Extraction and Analysis of Knowledge for Automatic Software Repair

Matias Martinez

Université Lille1 Sciences et Technologies

Département de formation doctorale en informatique
UFR IEEA

École doctorale SPI Lille

Extraction and Analysis of Knowledge for Automatic Software Repair

THÈSE

présentée et soutenue publiquement le 10/10/2014

pour l'obtention du

Doctorat de l'Université Lille 1

(spécialité informatique)

par

Matías Martínez

Composition du jury

Rapporteurs :	Yves Ledru, *Université de Grenoble 1* - France Tom Mens, *Université de Mons* - Belgique
Examinateurs :	Philippe Preux, *Université Lille 3* - France Diego Garbervetsky, *University of Buenos Aires* - Argentina
Directeurs :	Laurence Duchien, *Université Lille 1*- France Martin Monperrus, *Université Lille 1* - France

Laboratoire d'Informatique Fondamentale de Lille – UMR Lille 1/CNRS 8022 – Inria Lille - Nord Europe

Laboratoire d'Informatique Fondamentale de Lille

Mis en page avec la classe thloria.

Abstract

Bug fixing is a frequent activity in the software life cycle. The activity aims at removing the gap between *the expected behavior of a program* and *what it actually does*. This gap encompasses different anomalies such as the failure of a program facing to a given scenario. Bug fixing is a task historically done by software developers. However, in the recent years, several automatic software repair approaches have emerged to automatically synthesize bug fixes.

One of the main goals of automatic repair approaches is to reduce the cost of bug fixing compared to human fixing activity. Human fixing is a time consuming task for developers due to it involves doing manually task such as reproducing the bug, studying the symptoms, searching for a candidate repair and finally validating it. Software companies aim at decreasing the overall time of bug fixing and, by consequence, decreasing the money spent in the maintenance activity. For this purpose, researches and companies have proposed techniques and tools that help developers in particular activities such as debugging. Conversely, automatic software repair approaches emerge to provide a solution *from the beginning to the end*. That means, those approaches take as input a failing program and return a fixed version of that program.

Unfortunately, bug fixing could be even hard and expensive for automatic program repair approaches. For example, to repair a given bug, a repair technique could spend infinite time to find a fix among a large number of candidate fixes. Additionally, even the cost of fixing could be low, the cost of having a bug in production could be much higher. For instance, a bug could set a program off-line, or produce an economic damage due to its incorrect operational behavior. For that, finding faster a fix implies decreasing the time to release a fix and to deploy it in production. The main motivation of this thesis is to decrease the time to find a fix.

The proposed automatic repair approaches are evaluated by measuring the efficacy of repairing a set of defects. That means, given a defect dataset, the evaluation measures how many of them an approach is able to repair. For instance, in an evaluation of GenProg, a state-of-the-art repair approach guided by genetic programming, this approach repaired 55 out of 105 defects. A question that motivates this thesis is *What happens with the remaining (unrepairable) bugs from those repair approach evaluation experiments?* In particular, we wonder whether they are unrepairable due to: a) the repair search space, i.e., all possible solutions for the fix, is too large and the evaluated approach needs to spend long time to find one solution; or b) the approach is not able to fix these bugs i.e., it does not target repairing these kinds of bugs. For example, an approach that repairs defects in *if conditions* is not capable, in theory, of repairing memory leak defects.

In this thesis, we aim at finding answers to those questions. First, we concentrate on the study of *repair search spaces*. We aim at adding repair approach strategies to optimize the search of solutions in the repair search space. We present a strategy to reduce the *time* to find a fix. The strategy consumes information extracted from repairs done by developers. To obtain this information, we first extract source code repairs done by developers from open-

source projects. Then, we analyze the composition of those repairs, in particular, the changes done by each. Finally, we build change models that describe the kinds of changes and their abundance. The strategy uses these models to decrease the time to navigate the repair search space. It focuses on the most frequent kinds to repairs in order to find faster a solution.

We present a second strategy to reduce the *size* of *repair search spaces*. In particular, we focus on reducing the search space of one kind of automatic repair approach: *redundancy-based repair approach*. This kind of approach works under the assumption that the ingredients of a fix, i.e. the source code that conforms the fix, was already written before somewhere in the program. We call it *temporal redundant code*. We analyze the location of redundant code in the application. This study allows us to prove that it is possible to reduce the repair search space of redundancy-based approaches without losing repair strength. Both strategies are able to increase the repair efficacy of approaches, that means, to find solutions not previously found.

Then, we focus on the *evaluation of automatic repair approaches*. We aim at introducing methodologies and evaluation procedures to have meaningful repair approach evaluations. First, we define a methodology to define defect datasets that minimize the possibility of biased results. The way a dataset is built impacts on the result of an approach evaluation. We present a dataset that includes a particular kind of defect: if conditional defects. Then, we aim at measuring the repairability of this kind of defect by evaluating three state-of-the-art automatic software repair approaches. We carry out a meaningful evaluation of software repair approaches to determine whether: a) they are able to repair defects from the evaluation dataset; and b) which is better. We set up an experimental protocol where the number of free variables and potential biases are minimized. In summary, the experiments done in this thesis allow us to determine that it is possible to reduce the time of repair bugs and it is possible to carry out meaningful evaluations for measuring the real strength of automatic software repair approaches.

Contents

Chapter 1

Introduction

1.1 Context

The world is day by day more computerized. Software is everywhere: PC, laptops, tables, TV, video game stations, smart-phones and gadgets such as watches and glasses. Unfortunately, having new software means also having more and more new defects.

Bug fixing is an activity for removing defects in software programs. This activity aims at correcting the behavior of a program. It removes the gap between *the expected behavior of a program* and *what it actually does*. This gap encompasses different anomalies such as the failure of a program facing a given scenario. Examples of bug fix are: the addition of an *if precondition* to check whether a variable is not null before accessing it; the addition of a missing method invocation; or a change in an arithmetic operator.

When a bug is found, developers or application users report it in an issue tracking system. For instance, Apache Software Foundation[1], which provides support for more than 100 open-source software projects, uses the tracking system Jira[2]. Nowadays, companies such as Google or Facebook are worried[3] [4] of having bugs in their products. They do not want to loose clients or money. They have monetary rewards for the discovery of vulnerabilities to individuals, external to the company, who found legitimate issues. These companies aim at searching repairs for their bugs as soon as possible.

Bug fixing is a task historically done by software developers. Consequently, the increase of new bugs produces an increase in the demand of developers for fixing those bugs. For the industry, bug fixing has an economical impact: companies need to pay developers for fixing bugs. For open source projects, managed by not-profit organizations such as Apache or Mozilla, defects in their products also have a negative impact: their developers, volunteers that offer their time for free, have to spend much time fixing bugs. For instance, for Apache common Math library, a well-known library for math in Java environment, 28 days were the average time to repair each of the 52 bugs reported in 2013[5].

[1] http://www.apache.org/

[2] https://www.atlassian.com/software/jira/

[3] https://www.facebook.com/whitehat/

[4] https://www.google.fr/about/appsecurity/reward-program/

[5] Value obtained from the dashboard provided by the Apache issue tracker `https://issues.apache.org/`

To decrease the time of bug fixing (and the related economic cost), researchers and companies have proposed techniques and tools that help developers in particular development and maintenance activities such as debugging, or fault localizations. In spite of those contributions, developers continue spending effort on bug fixing.

As solution, in the recent years automatic software repair approaches have emerged to automatically synthesize bug fixes. One of their main goals is to decrease the time and economical cost of bug fixing. Furthermore, these approaches aim at providing a solution *from the beginning to the end*. That means, they take as input a failing program and return a fixed version of that program. Approaches such as GenProg [1], Patchika [2], ClearView [3], AutoFix-E [4] and PAR [5], have been proposed by the software engineering research community to fix real bugs.

Those automatic software repair approaches are able to automatically synthesize bug fixes. For that, they need two entities. One is the *bug oracle*: an entity that indicates the presence of a bug. The other is the *program correctness oracle*: an entity that indicates whether a program fulfills the software specification or not. The role of these oracles can be carried out by humans or by automatic entities such as a program. The former oracle corresponds to humans (user, developers, etc.) who decide whether a program works correctly or not. For instance, Carzaniga et al. [6] use an oracle to repair JavaScript bugs. After a repair is synthesized and integrated to a program, the user of the program under repair decides whether a synthesized repair fixes a bug (previously notified by the same user) or not. An automatic oracle has the structure of a function $f(P) = c$, where P is the program under repair, and c indicates whether the P contains a bug or not. This oracle has the advantage that repair approaches can use them in an automatically way. However, it is unusual and expensive to have encoded the complete specification of a program in one oracle. As solution, many approaches such as GenProg or Patchika rely on test suites as proxy to the ideal program specification. A program that passes all test cases from a test suite means it is correct according to its specification. Otherwise, if at least one test case fails, it means the program has a bug.

Repair approaches are evaluated to measure their strengths i.e., their repair capability. These evaluations usually contain two main phases. The first one is the *setup* of the evaluation process. The goal of this phase is to define a dataset of defects to be repaired by the approaches under evaluation. Defects can be collected from diverse sources: previous evaluations, defect repositories such as SIR [7], or from software projects (commercial or open-source). The defects can be real or artificially created defects. The second phase is the *execution* of the repair approach evaluation. A quantitative evaluation commonly consists on the measure of its *repair efficacy* over a dataset of defects. The repair efficacy measures the number of defects successfully fixed over the total number of defects from the evaluation dataset.

1.2 Problem

Repair approach evaluations from literature show that a fraction of evaluated defects remain unrepairable. An unrepairable defect means that the evaluated approach is not capable to find a bug fix. For example, GenProg is able to repair 55 out of 105 defects proposed in its evaluation [8]. Hence, it means that there remain 50 unrepairable defects in this dataset.

In this thesis, we wonder the reasons for having unrepairable defects in evaluations.
Is it a problem of the approach?
or
Is it a problem of the repair approach evaluation design?

We suspect both cases are possible. First, let us consider the case that an evaluated approach is not able to find a repair. A *search space* is the set of all *candidate solutions*, i.e., candidate repairs, that the repair approach is able to synthesize. Using the bug and correctness oracles, the approach determines whether a candidate repair is a solution (it repairs the bug) or not. A reason for not finding a solution could be the size of the search space that is too large and takes a long time to evaluate each candidate solution. Our intuition is there are sources of information that can be used by the approaches for improving the repairability of bugs. That means, these sources help approaches to find faster a solution in the repair space. As previous research shows [9], bug fixing is a repetitive task: most of the bugs can be fixed using a finite set of bug fix patterns i.e., common fixes. Few repair approaches from the literature use knowledge from previous human bug fixes. The state-of-the art approach PAR [5] is one that partially uses it. It synthesizes fixes by instantiating 10 bug fix patterns (recurring similar patches) derived from open-source bug fixes. However, there is more valuable knowledge from the software repositories to increase the repairability strength of repair approaches. For instance, despite that the work of Pan et al. [9] presents 27 bug fix patterns, only a small portion of this knowledge is used in the automatic software repair field.

Second, we suspect that evaluation procedure is not well designed and, by consequence, the evaluation does not produce accurate results. Let us present an illustrative example of a not well-defined evaluation. Consider that we want to evaluate one language translator, that translates words from English to Spanish. The evaluation corpus is a text written half in English and half in Chinese. The translated text (the translator output) ideally would have half of the words in Spanish, half in Chinese. A well-defined evaluation should only consider words that the evaluated translator targets, in this example, English words. Then, the evaluation can calculate measures such as efficacy (number of words corrected translated over all words evaluated from the target language). Contrary, a not well-designed evaluation, such as that one from our example, includes words from a language not targeted by the translator under evaluation. The words of the not-targeted language (in the example, Chinese words) introduce noise in the evaluation. The evaluation result does not show the real efficacy of the translator. This example could sound a bit naive. However, in a repair approach evaluation, a not well-defined evaluation process could produce the same effect.

Now, let us focus on evaluations of repair approaches. We claim that repair approaches always target defect classes. A *defect class* is a family of defects that have something in common such as the root cause or the kind of fix applied to repair the defect. In this context, an approach targeting a defect class means it is able to repair defects of that class. For example, the repair approach Semfix [10] is able to repair bugs in *if conditions* but not *missing method invocations*. Coming back to the automatic translation example, approaches are the translators, and the defect classes are the languages that a translator targets (e.g., English to Spanish). Having this information, one is able to characterize evaluation datasets. We have two kinds of defects. On the one hand, the *repairable defects*, which can be repaired by the evaluated repair approach. The repair approach targets the defect classes of repairable defects. On the other hand, the *unrepairable defects* are the defects that cannot be repaired, by definition, by the evaluated repair approach. That is, the repair approach does not target the

defect classes of the unrepairable defects and, by consequence, its repair search space does not contain any solution. In the previous example, English words are equivalent to *repairable defects* and Chinese words to *unrepairable defects*. The result of the evaluation would not be the same if a dataset contains more repairable bugs than unrepairable, and vice-versa.

To summarize, the problem we found in the evaluations from the literature is that they present a portion of defects used in the evaluation that remains unrepairable. Moreover, those evaluations do not always include the inclusion criteria of the defect dataset used. As consequence, it is not possible to have a sound measure of an approach's performance. In particular, it is not possible to determine whether a defect could not be repaired due to the intrinsic characteristics of the repair approach or whether the evaluation process is not appropriate.

1.3 Thesis Contribution

Our main objective is to improve the performance of repair approaches. That means, to increase the number of solutions for *repairable defects* from the evaluation dataset. To measure the performance of a repair approach, we also need to conduct meaningful repair approach evaluations. That involves characterizing the defect dataset used in the repair approach evaluations. These two objectives are guided by the following hypotheses:

a) The repairability of defects can be improved by considering knowledge from previous repairs done by human developers.
b) Evaluation results are meaningful if evaluation uses well-defined datasets.

In this thesis we validate both hypotheses. For the first one, we search for strategies for discovering repairable defects that are not previously found by an existing approach. These strategies rely on information from previous repairs and already written source code. We aim at adding those strategies to existing repair approaches to optimize the search of solutions in the repair search space.

We first present a strategy that uses information from previous repairs to optimize the search of solutions in the *repair search space*. The optimization reduces the *time* to find a fix. The strategy focuses on the most frequent repairs in order to find a solution faster. For that, it consumes information extracted from repairs done by developers. We mine version control systems (VCS) of open-source projects to extract those repairs. Version control systems are used by developers to track changes done during the software life-cycle. Then, we analyze the composition of those extracted repairs. Using this information, we build a model that captures the source code changes used in bug fixing activity and their abundance. The strategy uses these models to decrease the time to navigate the repair search space. A repair approach that implements this strategy aims at synthesizing fixes composed by the most frequent bug fix changes. This thesis is the first to study the incorporation of this knowledge in the context of automatic software repair.

Then, we present a novel approach to identify repairs from version control systems. We present a method to specify *patterns of source code changes*, and an approach to identify instances of those patterns in a software history. The combination of both allows us to extract knowledge from software history. We use our approaches to specify bug fix patterns from the Pan et al.'s bug fix patterns catalog [9], and then we measure the abundance of each of

them. This knowledge could be applied to improve the repairability of *bug fix pattern-based repair approaches* such as PAR [5].

We present a second strategy to optimize the search of solutions in the *repair search space*. It reduces the *size* of repair search spaces without significantly decreasing their number of solutions. Reducing the size has a direct impact on the repair time. We focus on reducing the search space of one kind of automatic repair approach: *redundancy-based repair approach*. These approaches work under the assumption that the ingredients of a fix, i.e., the source code that conforms the fix, were already written before somewhere in the program. We analyze the location of redundant code in the software. This study allows us to prove that it is possible to reduce the repair search space of redundancy-based approaches without losing repair strength.

Our second hypothesis states that when the evaluation procedure is not well designed, the evaluation result could not reflect the real strength of the evaluated approach. That means, for instance, the repair efficacy measure could be high due to a biased definition of the evaluation dataset. To validate our second hypothesis, we study the dimensions of evaluation for repair approaches.

First, we focus on the setup step of an approach evaluation. Our challenge is to define a methodology to build unbiased evaluation datasets. For that, it is necessary to characterize how the dataset is built and what it contains. As we have shown in the automatic translation example, a biased evaluation dataset impacts on the result of an evaluation. Suppose one wants to evaluate an approach that is able to repair defects of class A, but not defects of class B. If one considers a dataset including a majority of A defects, the efficacy of the approach can possibly be high. Contrary, if one includes a majority of B defects, the efficacy would be low, by construction. The reason of these opposite results is the presence of bias in the evaluation dataset (in this case the presence of unrepairable defects). This bias could be unintentional, especially when the dataset is informally built without well-defined criteria. As consequence, measures such as *repair efficacy* depend on the conjunction of the following factors: *a*) the definition of the dataset (the defect classes included in the dataset and their abundance); and *b*) the capacity of the approach to repair a given defect class (e.g., the illegal access to an uninitialized variable).

Those factors have a great impact on the conclusiveness for the evaluation of a repair approach. Unfortunately, evaluations of repair approaches from the literature do not include the criteria used to define evaluation datasets. Thus, the evaluation result could be biased. For example, in the evaluation of PAR [5], the authors define a dataset of 119 defects taken from the issue tracking system of six open source projects. The inclusion criterion of those defects is not well-defined. The authors state that "we randomly selected 15 to 29 bugs per project". Moreover, it is missing the notion of defect classes. The evaluation returns that PAR fixes more defects that GenProg. We think that using a strong inclusion criterion, the result could eventually be different and would better characterize the strength of repair approaches. Our methodology aims at obtaining this result.

Finally, we focus on the execution step of a repair evaluation. Our motivation is to execute evaluations of repair approaches that produce reliable and conclusive results. This experiment allows us to concretely instantiate our evaluation methodology in order to validate it. In particular we aim at studying the repairability of a particular defect class: *if condition* defects. These defects are common: previous works [9, 11] have shown that they are the most repaired elements in source code. According to the results of Pan et al. [9] over

six open source projects, between 5% and 18.6% of bug fix commits are modifications done in *if conditions*. Through our experiment we also want to know: *a*) whether *if conditions* are automatically repairable; *b*) whether one of the major repair approaches of the literature is better than the others on this defect class. We consider three repair techniques from the most authoritative literature: GenProg [12], PAR [5] and the mutation-based approach defined by Debroy and Wong [13]. To carry out this experiment, we define an experimental protocol where the potential experimental biases are minimized. We propose a unified repair framework to reproduce the three repair approaches. The framework factorizes the variabilities of the three approaches under consideration and allows us to implement the particular behavior of each of them. For instance, the framework uses, for the three implementations, the same mechanism to detect the suspicious buggy locations and the same correctness oracle mechanism to validate candidate fixes.

1.4 Outline

The remained of this thesis is structured as follows. Chapter 2 provides an overview of previous work that focuses on the analysis of software evolution and automatic software repair approaches. Chapter 3 provides an analysis of human repairs mined from source code repositories. Chapter 4 presents two strategies to optimize search-based software repair. Chapter 5 presents a framework for evaluation of repair approaches. Chapter 6 presents the summary of the thesis and future works.

1.5 Publications

Published:

M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, (New York, NY, USA), pp. 492–495, ACM, 2014.

M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Engineering*, pp. 1–30, 2013.

M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with ast analysis," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 388–391, Sept 2013.

To appear:

J.R Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus "Fine-grained and Accurate Source Code Differencing". To appear in *Proceedings of the 2014 29th IEEE/ACM International Conference on Automated Software Engineering*, ASE '14.

Technical Report:

M. Monperrus and M. Martinez, "CVS-Vintage: A Dataset of 14 CVS Repositories of Java Software," tech. rep `http://hal.archives-ouvertes.fr/hal-00769121`.

To be submitted:

M. Martinez, M. Monperrus, "Automatically Fixing Buggy If Conditions: an Empirical Evaluation of Three Software Repair Techniques",2014

M. Martinez, L. Duchien, and M. Monperrus, "Accurate extraction of bug fix pattern instances using abstract syntax tree analysis," 2014.

Chapter 2

State of the Art

In this thesis we aim at extracting knowledge from repairs done by developers. We want to use this knowledge to improve the performance of automatic software repair approaches. In this chapter we present the prior work relevant to our thesis. First, in Section 2.1 we introduce the definitions of relevant concepts used along this thesis. Then, in Section 2.2 we analyze the related works on software evolution and bug fixing activity. Finally, in Section 2.3 we focus on presenting proposed automatic software repairs and their evaluations.

2.1 Definitions

In this section we present the list of widely used terms along this thesis. This list is presented in alphabetic order.

An **automatic software repair approach** is a software that receives a program with one defect and automatically produces a repair that solves this defect.

A **biased dataset** is a collection of defects defined through a biased sample.

A **biased sample** is a subset of a population (i.e., defects) that is not representative of —either intentionally or unintentionally—an entire population.

A **bug oracle** is an abstract entity that decides whether a program has a bug or not.

A **bug fix commit** is a commit containing changes to fix buggy code.

A **bug fix pattern** is a set of source code changes that frequently appear together for fixing a bug.

A **change action** represents a kind of source code modification. A change action is part of a **change model**.

A **change model** represents a feature space of software changes. For instance, the change model of standard Unix diff is composed of two changes: line addition and line deletion. For instance, in a **probabilistic change model** each change action has associated a value with a probability that this action occurs.

A **change pattern** is a set of source code changes that frequently appear together.

A **change pattern identifier** is a software entity that classifies, if it is possible, a concrete change as an instance of a change pattern.

A **correctness oracle** is an abstract entity that decides whether a program is correct according to its specification or it is invalid.

A **defect class** is a family of defects that have something in common such as the root cause or the kind of fix.

A **defect dataset** is a set of defects used for experiments and evaluations. For instance, in the automatic software repair domain, defect datasets are used to measure the performance of automatic software repair approaches.

An **experimental bias** is a process where the scientists performing the research influence the results, in order to portray a certain outcome[6]. Bias in research occurs when systematic errors are introduced into sampling or testing by selecting or encouraging one outcome or answer over others[7].

A **hunk** is a set of co-localized changes in a source file. In lines of code, a hunk is composed of a consecutive sequence of added, removed and modified lines.

A **hunk pair** is a pair of related hunks, one hunk being a section of code at version n and the other being the corresponding code in the fixed version $n+1$. Hunks are paired through the process of differencing that computes them.

A **repair action** is a change action that often occurs for repairing software, i.e., often used for fixing bugs.

A **repair model** is a set of repair actions, i.e., a set of source code changes often used for fixing bugs.

A **repair search space** is a space of all candidate solutions (i.e., fixes) that an automatic software repair is able to generate. A candidate solution must be validated to verify whether it is a valid fix or not.

A **repair synthesis** is the process for generating source code of a repair.

A **revision** is a set of source code changes done over one file and committed in the version control system. The revision produces a new *version* of the modified file.

A **versioning transaction** or **commit** is an atomic operation done by developers to reflect their working changes in a version control system. These changes produce a new *version* of the modified files.

A **version control system (VCS)** is a system that records the history of software changes during the development and maintenance of a software system.

2.2 Studying Software Evolution

In this thesis we aim at studying how bugs are fixed along the software life-cycle. Bug fixing activity is part of the software evolution. Our goal is to extract valuable information from bug fixing activity to apply in automatic software repair domain.

[6]https://explorable.com/research-bias
[7]http://www.merriam-webster.com/dictionary/bias

In this Section, we present works that focus on how software evolves through time. First, in subsection 2.2.1 we focus on those works that study the evolution at the source code level. That means, how source code changes through the time. We present works that compute the changes at the source code level. These works are used to compute, for instance, the changes between two consecutive versions of one file. We also analyze previous work that measure the abundance of change types, and others that detect change patterns i.e., set of source code changes that frequently appear together.

In this thesis we are interested in detecting the portion of software changes related to the bug fixing activity. Version control systems (VCS) are used by developers to track changes done during the software life-cycle. Developers introduce changes in the software through *commits*. In section 2.2.2 we present works that analyze commits from version control systems. Previous works have presented infrastructure to facilitate the study of the software evolution, i.e., the software history. In section 2.2.3 we present those state-of-the-art works.

Finally, once we are able to identify the portion of the software evolution related to bug fixing activity, we aim at knowing how this activity is done. For instance, what are the source code changes done to fix the most frequent bugs. In section 2.2.4 we present previous works that focus on bugs and fixes.

2.2.1 Studying How Source Code Evolves

In this section we present publications that analyze the software evolution at the *source code level*. First, in subsection 2.2.1.1 we present works that detect the software changes at the mentioned level. Then, in subsection 2.2.1.2 we present works that focus on measuring the abundance of those software changes. In subsection 2.2.1.3 we present works that identify *change patterns* i.e., set of source code changes that frequently appear together. Subsection 2.2.1.4 present works that define methods to specify patterns. Finally, in subsection 2.2.1.5 we identify the previous works about repetitiveness of source code.

2.2.1.1 Analyzing Source Code Changes

We are interested in studying how the source code changes along the software history. For instance, we want to calculate the source code changes introduced by a new version of a file. Our goal is to study the kinds and abundance of source code changes related to bug fixing activity.

Previous works focus on the computation of source code changes. The changes are expressed at a given granularity. For instance, at line granularity level (e.g., 1 line added) or at the abstract syntax level (AST) node granularity level (e.g., 1 AST node updated). In the context of software evolution, these approaches are used for comparing a version of a file with its predecessor.

First approaches calculate textual changes. They are usually based on the longest common subsequence algorithm. A well-know algorithm is Myers' differencing algorithm [14]. They compare two files and highlight elements (e.g., lines, chars) added and removed between them. These algorithms are not able to identify whether the change affects source code or documentation, or to identify the source code entity (e.g., method invocation, assignment) affected by the change.

To overcome these limitations, researches have proposed approaches based on, among other solutions, UML [15, 16], token [17] and AST [18, 19, 20, 21]. Let us focus on one of them,

which we extensively use in this thesis. ChangeDistiller [20] is a fine-grain AST differencing tool for Java. It expresses fine granularity source code changes using a taxonomy of 41 source change types [22], such as "statement insertion" of "if conditional change". ChangeDistiller handles changes that are specific to object-oriented elements such as "field addition". Moreover, Fluri et al. have published an open-source stable and reusable implementation of their algorithm for analyzing AST changes of Java code.

2.2.1.2 Measuring Frequency of Change Types

Once we are able to compute changes between two files (for example, two consecutive versions of a same file) at different level of granularity, we aim at measuring the abundance of each change type to know the importance of each.

Previous works studied the abundance of change types in the version control system of applications. For instance, Raghavan et al. [18] analyze patches obtained from a source code repository and then identify higher-level program changes. Their work shows the six most common types of changes for the Apache web server and the GCC compiler, such as "Altering existing function bodies", "Altered existing if conditions" and "Altered existing function calls". At a finer grain level, Fluri et al. [20] presented some frequency numbers of their change types [22] in order to validate the accuracy and the runtime performance of their distilling algorithm.

Nguyen et al. [23] present an empirical study of repetitiveness of code changes. They measure the repetitiveness of fine grained source code changes in general changes (considering all revisions form software repository) and bug fix changes. Their experiment return that "method calls", and "expressions" have the most number of changes, while changes to "constructor calls", and "do statements" are less.

2.2.1.3 Discovering Change Patterns

In this thesis we consider *change pattern* as a set of change types (such as addition of method invocation, update assignment) that appear frequently together. In this thesis we aim at studying change patterns related to bug fixing activity: the *bug fix patterns*. That is, we want to know how source code changes are frequently combined to fix a particular defect class.

Studies have focused on the extraction of change patterns from the history of the software. For instance, Fluri et al. [24] use hierarchical clustering of source code changes (from the taxonomy defined by ChangeDistiller) to discover source code change patterns. Livshits and Zimmermann [25] presented an approach to detect error patterns of application-specific coding rules. The proposed approach extracts automatically likely error patterns by mining software revision histories and checking them dynamically.

The mentioned publications, as many others in the empirical software engineering community, mine valuable information from version control systems (VCS) such as CVS, SVN or GIT. Version control systems are used by developers to track changes done during the software life-cycle. However, much code evolution data is not stored those systems [26, 27, 28, 29]. These systems store changes that commits introduce. However, the changes that developers do between two commits are not registed in those systems. This involves that they register a fraction of the software evolution data. As solution, approaches such as CodingTracker [26] or SpyWare [29] have been proposed to record fine-grained and diverse data

during code development. The authors of CodingTracker found that 37% of code changes are shadowed by other changes, and are not stored in VCS.

2.2.1.4 Pattern Formalization

In this thesis we aim at discovering *instances* of change patterns. For example, we want to collect the commits that introduce source code changes related to a particular change pattern. This would allow us to measure the importance of change patterns in the software history. For that, it is necessary to define a mechanism to formalize change patterns, and then another mechanism for searching instances from that formalization.

Previous works have focused on defining specifications of *patterns*. For example, Mens and Tourwé [30] presented a declarative framework for specifying *design patterns*, their constraints, and their high-level evolution transformations. Moreover, their framework allows defining refactoring transformations that can be applied to a given design pattern instance. The authors used their approach to specify design pattern from Gamma et al. [31] catalog such as the *Abstract Factory* design pattern. Beside this work, to our knowledge, no previous publications have focused on specifying a formalization of *change patterns*, i.e., a structure to define source code change pattern.

2.2.1.5 Study of Repetitiveness of Source Code

Previous work has focused on the study of code that, in the moment it was written, already existed somewhere in the application. We call this concept *software redundancy*.

Some works on code clone detection study software repetitiveness at line-granularity [32] and at token-level granularity [33]. For instance, Kim et al. [34] considered code clones via a temporal perspective, linking clones together across versions; Li et al. [35] presented a token-based approach to detect copy-paste bugs.

Other works have focus on the naturalness of software. Gabel and Su [36] studied the uniqueness of source code. Their *syntactic redundancy* calculates the degree to which portions of software applications are redundant. Hindle et al. [37] studied the repetitiveness and predictability of code. They present a statistical language model to capture the high-level statistical regularity that exist in code, represented by n-gram level, probabilistic chains of tokens. Both consider software redundancy from a spatial viewpoint.

2.2.1.6 Conclusion

The main lesson of this subsection is there is no previous work that defines a method to specify *change patterns* in a finer-grained manner. We aim at defining an approach to specify patterns. Moreover, we want to mine instances of these formalized patterns from the history of the software.

2.2.2 Empirical Studies of Commits

Developers introduce source code changes in version control systems through *commits*. Commits introduce modifications to existing software artifacts (source code files, configuration files, etc.), introduce new artifacts, and remove existing ones. Additionally, each commit contains meta-data such as the commit date, the name of the developer that does the commit, and a message log (a text where a developer can explain the purpose of the commit).

In this thesis we aim at studying commits that introduce bug fixes to know how developers repair bugs. Several works have studied the evolution of software at the level of version control systems *commits*.

In subsection 2.2.2.1 we present the works that focus on commits meta-data and metrics. Then, in subsection 2.2.2.2 and subsection 2.2.2.3 we present works that focus on commits that introduce fix and bugs, respectively.

2.2.2.1 Studying Commit Meta-data and Metrics

Some works focus on commit meta-data (e.g. authorship, commit text) or size metrics (number of changer files, number of hunks, etc.). For example, Alali et al. [38] discussed the relations between different size metrics for commits (# of files, LOC and # of hunks), along the same line as Hattori and Lanza [39] who also consider the relationship between commit keywords and engineering activities. German [40] asked different research questions on what he calls "modification requests" (small improvements or bug fix), in particular with respect to authorship and change coupling (files that are often changed together). On the opposite, Hindle et al. [41, 42] focus on large commits, to determine whether they reflect specific engineering activities such as license modifications. Other works analyze the relation between commit meta-data and the evolution of the source code. For instance, Fluri et al. [43, 44] look at the relation between source code and comment changes. They found that when code and comments co-evolve, both are changed in the same revision: 97% of comment changes are done in the same revision as the associated source code change.

2.2.2.2 Searching for Bug Fixes Commits

Previous researches have focused on the identification from the software history of: a) changes (instances) that fix bugs and b) changes that introduce bugs. For both cases, techniques propose to identify commits that introduce those changes. One approach presented by Mockus and Votta [45] identifies keywords in the commit message. In their work, they classify a change as "corrective" if the message log of the commit that introduces the change contains one of the following keywords: "fix", "bug", "error". Other techniques search links to external system in the commit message. The idea is to relate commits with reports such as a bug report from issue tracking systems. The linkage technique is a way to combine two different sources of information. For example, techniques such that one presented by Fischer et al. [46] discover links between commits and issue reports, explicitly written in the commit message.

Other works have empirical studied these linking techniques. For instance, Bird et al. [47] found these heuristics could produce bias results due to developers can omit bug reference (the link) in the commit message. The authors found that 54% of fixed bugs in the bug dataset are not linked to commit message (missing links). Moreover, Antoniol et al. [48] studied linked reports from issue tracker and obtain that less than half of them related to corrective maintenance (bug fixing), while the rest were related to activities such as enhancements or refactoring. Some approaches have emerged to discover missing links. Wu et al. [49] presented an approach called ReLink to automatically recover links. ReLink learns criteria of features from explicit links to recover missing links. Another approach [50] uses machine learning approach for text categorization of fixing-issue commits on CVS, while

[51] used a probabilistic approach to effectively recover traceability links between bug fix commits and corresponding bug reports.

2.2.2.3 Searching for Bug-introducing Commits

Other works have focused on the study of commits that introduce bugs [52, 53]. A fix-inducing change is a change that later gets undone by a fix. As Kim et al. [53] state, the extraction of bug-introducing changes is challenging, in contrast to bug-fixes that are relatively easy to obtain. In addition to the use to linkage technique to detect a fix commit, these approaches define algorithms of find the commit that introduces the bug. Those algorithms are based on line diff analysis [52] or annotation graphs [53, 54]. Bug-introducing changes also were studied by Purushothaman and Perry [55]. They studied small commits (in terms of number of lines of code) of proprietary software at Lucent Technology. They showed the impact of small commits with respect to introducing new bugs, and whether they are oriented toward corrective, perfective or adaptive maintenance. Filtering bug fix commits allows us to study the features of kind of commits. For instance, the common source code changes they introduce.

2.2.2.4 Conclusion

In this section we learned that previous works define methods to collect the portion of the software evolution related to bug fixing activity. Some of them rely on mining commit's message logs and other on commit's features such as commit size. In this thesis we aim at studying the relation between bug fixing commits and commit size in terms of AST changes affected by the commit.

2.2.3 Defining Infrastructures for Software Evolution Analysis

As Mens et al. state [56] one of the challenge of software evolution is to find out how different kinds of data (bug reports, change requests, versioning repositories, execution traces, etc.) can be integrated, and how support this integration can be provided. Previous works defined infrastructures to facilitating software evolution research such as Evolizer [57], Kenyon [58], Hipikat [59], RHDB [46], SoftChange [60]. These infrastructures group information from different systems used in development such as version control systems, issue trackers or mailing lists. They also provide mechanisms to querying the data. For instance, Fischer [46] defined an infrastructure built from basic building blocks (SQL database and scripts) for retrieval and filtering information from a) the version control system and b) the bug report database. Likewise, SoftChange [60] retrieves, summarizes, and validates data from mailing lists, CVS logs, and defect tracking systems based on Bugzilla.

2.2.4 Study of Bugs and Fixes

Once we detect commits from version control systems that introduce fixes, we aim at studying what these fixes look like. Previous works characterize bugs and fixes. In this section we first focus on works that present bugs and fixes classification (subsection 2.2.4.1), and then works that analyze the abundance of bug and fixes (subsection 2.2.4.2).

2.2.4.1 Taxonomies

In this thesis we are interested in how to classify bugs. To repair defects and propose bug fixes, we need first to know: a) What bugs look like. In particular, the different types of bugs, the places they affect, their manifestation; b) the changes that developers do to fix bugs.

In one hand, previous works from the literature classify software defects. The preliminary works manually define a classification from, for example, bug reports or software documentation. For instance, Knuth defines a schema to classify the errors of TEX from his error log [61]. Chillarege et al. [62] present an Orthogonal Defect Classification (ODC). It corresponds to a categorization of defects into classes called *Defect Types*. This categorization has eight defect types such as Assignment (error in initialization of control blocks or data structure), Interface (errors in interacting with other components) or Documentation error. Moreover, Ostrand and Weyuker [63] present a scheme for categorizing software errors. They characterized an error in distinct areas, including "major category", "type", presence, and use of data. For example, "major category" identifies what type of code was changed to fix the error. They develop this classification schema from change reports filled out by developers of an industry software product. Then, they present the number and percentage of errors for each area.

In the other hand, previous works concentrate on the analysis of bug fixes. For instance, Pan et al. [9] manually identified 27 bug fix patterns on Java software from previous bug fixes done by developers. A bug fix pattern is set of common source code changes applied to fix a particular kind of bug. The authors have manually analyzed bug fix commits from open source commits to define the pattern catalog. Moreover, Nath et al. [64] have extended this bug fix pattern catalog adding three alternative patterns discovered from this manual inspection. There are approaches that automatically learn project-specific bug fix patterns from software versioning history [65, 66].

2.2.4.2 Measuring Abundance of Defect and Fixes

In this thesis, we aim to knowing what source code changes are frequently used in bug fixing activity. Our intuition is this information would allow us to improve the performance of automatic software repair approaches. This improvement can be done by first concentrating on synthesizing repairs frequently done by developers.

Previous work focuses on the measurement of the abundance of each element defined in those bug or fix taxonomies. This measure is done by analyzing software artifacts such as reports [63], or source code [67, 9, 64, 68]. For instance, Pan et al. [9] present a tool to extract instances of the 27 bug fix patterns of their catalog from SVN version control repositories. Contrary, Nath et al. [64] replicate the experiment by manually measuring the abundance of 27 Pan et al.'s pattern plus the pattern they introduced. Both works also shows that there is a portion of bug fix commits in the software history (detected by the method explained in Section 2.2.2.2) that do not contain any instance of the defined bug fix patterns. That means, it could exist more unknown bug fix patterns.

2.2.4.3 Conclusion

The main lessons that this subsection gave us are twofold. First, there are bug fix pattern not already discovered or formalized. Second, the replication of experiments that measure

the abundance of pattern instance is difficult. The main reasons are: few tools are defined and available, and these tools are not flexible to accept new patterns. In this thesis we aim at measuring in a flexible way: independent of the pattern to search. We want to define an approach that is able to measure the abundance of change pattern already defined and new one as well.

2.3 Automatic Software Repair

Automatic software repair approaches have emerged to provide a patch (at source code level or binary level) that fixes a software defect. Researchers have modeled the synthesis of patches as a search problem [69, 70]. Search-based software engineering seeks to reformulate software engineering problems as search-based problems [70]. It aims at applying meta-heuristic search-based techniques such as genetic algorithms [71]. A search space contains all candidate solutions. In the automatic software repair context, a solution is a patch.

One of the main problems in automatic software repair is to have an oracle that indicates whether a program is *correct* according to its specification (i.e., it does not contain bugs). Automatic software repair approaches need those oracles to determine whether a program contains a defect, and for a buggy program, to determine whether a candidate solution, chosen from the search space, is a solution (i.e., fixes the bug) or not. Unfortunately, a program specification is not always explicit or accessible in an automatic way. As consequence, a challenge of software repair approaches is also to define those oracles. Usually, requirement engineers and/or developers specify the behavior program in documents written in natural language and using modeling language such as UML [72]. However, this natural representation is a barrier for automatic repair approaches. Paradigms such as design-by-contract programming (DBC) [73] include formal specification in the software. Bertrand Meyer developed DBC as part of his Eiffel programming language. For instance, it uses preconditions and postconditions to document (or programmatically assert) the change in state caused by a piece of a program. Repair approaches such as AutoFix-E [74, 4] leverage on this kind of specification to repair bug fixes. However, this contract does not supply the correctness oracle.

Researchers on automatic software repair domain have relied on *test suite* as a proxy of software specification [75, 12]. A test suite is a collection of test cases used to test a software program to verify whether it fulfills some specified behaviors. A test suite presents as advantage that it can be executed in an automatic way.

In this thesis we focus on a particular kind of repair approaches: *Test suite-based repair approaches*. These approaches rely on test suite as bug and correctness oracles. If at least one test case fails means the program under evaluation has a bug and it does not fulfill its specification. Moreover, these approaches verify the whether a candidate patch is valid or not. A synthesis patch must pass the original failing test cases, and must keep the other test cases passing.

We aim at improving the repairability of test suite-based repair approaches. Our goal is to define strategies that allow existing repair approaches to repair defects that were hard to repair. We also want to carry out meaningful evaluation of repair approaches. The evaluation of repair approach takes defect from a defect datasets and tries to repair each of them with the approach under evaluation.

In subsection 2.3.1 we present test suite-based repair approaches from the literature.

Then, in subsection 2.3.2 we present strategies to optimize the repair time. Finally, in subsection 2.3.3 we present works that define defect datasets.

2.3.1 Test Suite-based Repair Approaches

Previous works have defined test suite-based repair approaches based on search-based optimization techniques. Arcuri presents JAFF [69], an approach that uses search-based techniques (hill climbing and genetic programming) to repair software. Weimer et al. [12] introduce GenProg, a genetic programming approach to C program repair. The approach defines genetic operations that use existing code from other parts of the program to synthesize patches. That means, GenProg never introduces new code into the application. As difference with the mentioned work presented by Arcuri, GenProg was validated using real defects from large programs. In the last evaluation of GenProg [8], the approach fixed 55 out of 105 bugs. Another approach leverages evolutionary computing techniques to generate program patches is PAR [5]. It generates program patches automatically from a set of 10 manually written fix templates. PAR synthesizes fixes by instantiating those bug fix templates and, for some of those, PAR does it taking by existing code from the same program. The evaluation of PAR returns that the approach fixed 27 out of 119 bugs from 6 open source projects.

Qi et al. [76] present RSRepair, an approach that tries to repair faulty programs with the same operators as GenProg. RSRepair uses random search to guide the process of repair synthesis rather than genetic programming. Their evaluation shows that RSRepair, in most cases, outperforms GenProg in terms of both repair effectiveness (requiring fewer patch trials) and efficiency (requiring fewer test case executions).

Dallmeier et al. [77] present a repair approach named Patchika that a) infers an object usage model from executions, b) determines differences between passing and failing runs, and c) generates fixes that alter the failing run to match the behavior from the passing run. Patchika automatically builds finite-state behavioral models for a set of passing and failing test cases of a Java class. Then, it can insert new transitions or delete existing transitions to change the behavior of the failing model. Patchika was able to fix 3 out of 18 bugs from an open source project. Wei et al. [74] present AutoFix-E, an automated repair tool which works with software contracts. In particular, it repairs Eiffel classes, which are equipped with contracts (preconditions, postconditions, intermediate assertions, and class invariants). Autofix shares the same foundation with Pachika such as the use of behavior models, state abstraction, and creating fixes from transitions. AutoFix-E leverages user-provided contracts integrating them with dynamically inferred information. In its evaluation, AutoFix-E repaired 16 out of 42 bugs from two Eiffel libraries.

Inspired from the field of mutation testing, Debroy et al. [13] present an approach to repair bugs using mutations. For a given location, it applies mutations, producing mutants of the program. A mutant is classified as "fixed" if it passes the test suite of the program. Their repair actions are composed of mutations of arithmetic, relational, logical, and assignment operators.

Using semantic analysis, SemFix [10] approach explicitly focuses on if conditional defects. It generates repairs by combining symbolic execution, constraint solving, and program synthesis.

2.3.2 Optimizing Repair Runtime

Previous works propose extensions or modifications of existing approaches to optimize the search of the solution. These optimizations aim at repairing faster and, by consequence, to increase the repairability strength of an approach

AE [78] is an extension of GenProg that aims at decreasing both repair time and repair cost compared with GenProg by analyzing equivalent programs and applying test case execution reduction. The approach applies the same source code operators as GenProg, but the solution search is not guided by genetic programming.

Some automatic software repair approaches such as GenProg use evolutionary computing to generate candidate patches. One drawback is the time cost. Qi et al. [79, 80] show that weak recompilation can reduce the time cost of repairing. Weak recompilation is only compiling and installing the changed source code, without dealing with unchanged source code. This approach can avoid the time cost of unchanged source code in multiple patches. The time cost (in seconds) shows that weak recompilation can reduce at least $4/5$ time cost in large programs (namely, Php and Wireshark). Note that no accuracy is compared since the technique of generating patches is the same.

Another optimization in program repair is done through fault-recorded testing prioritization [81]. *TrpAutoRepair* is an extension of GenProg that aims at reducing the number of test case executions in the repair process. The evaluation shows that the approach can significantly improve the repair efficacy by reducing efficiently the test case executions when searching a valid patch in the repair process.

Those optimization strategies focus on specific stages of the repair process such as fault localization [82, 83], repair synthesis [79] or candidate repair validation [81]. However, no approach focuses on the strategy of selecting the kind of repair to apply in a buggy location. In this thesis we present one strategy using information from previous fixes to select. The strategy aims at selecting first the most common repair shapes to decrease the repair runtime.

2.3.3 Bug Benchmarks and Datasets for Evaluation of Repair Approaches

In this section we present related works that define defect datasets and benchmarks. There is a large number of previous works that focus on static fault localization [84, 85]. FaultBench [86] provides a benchmark for evaluation and comparison of techniques that prioritize and classify alerts generated by static analysis tools.

Benchmarks such as BugBench [87] and BegBunch [88] are defined for evaluating bug detection tools. Both benchmarks contain bugs in C/C++ code and more than one type of bugs, for example, memory leak or buffer overflow bugs. Lu et al. [87] present a guidelines on the criteria for selecting representative bug benchmark.

In this thesis we focus on evaluation of *test-suite based program repair approaches*. These approaches use a test suite to validate the correctness of a program and, by consequence, to know whether a program has a defect or not. So that, to evaluate these approaches, the program under repair must include a test suite that validates its correctness and exposes the defect (at least one failing test case).

Benchmarks from the literature include defects from programs with test suites. For instance, Do et al. [7] present SIR, an artifact repository that includes versions of Java, C, C++ and C# programs with defects. SIR includes real and seeded defects. In our opinion, seeded bugs produce a bias in the result of approaches evaluation. A dataset with seeded

bugs is biased by the defect classes seeded, that were artificially synthesized. These defect classes usually are a subset of all defect classes existing in software. Additionally, the distribution of each kind of defect seeded could be different from its distribution in real programs. Dallmeier et al. [89] present iBugs, a technique to automatically extract bug localization benchmarks from a project's history. Additionally, they present and publish a publicly available repository containing 369 bugs. They recognize bugs and fixes from commits by analyzing the commit's meta-data i.e., commit message.

2.3.4 Conclusion

The main lesson we learned from this subsection is that neither existing repair approaches nor optimization techniques consider information from previous repairs. Moreover, we learned that nobody has defined datasets for evaluation test-based repair approaches, with a well-defined built criteria such as the defect classes it contains. In this thesis we aim at including information in repair strategies to decrease the repair time. Moreover, we want to define meaningful defect datasets with explicit built definition for evaluating test-based repair approaches.

2.4 Summary

In this chapter we studied state of the art related from domains such as test suite-based program repair approaches and analysis of software evolution. Moreover, we have detect their limitations and opportunities to extend and improve them.

First, we observe that the majority of state of the art approaches presented in this section does not profit from any information of previous repairs done by developers. The exception is PAR repair approach, which partially uses 10 bug fix templates manually mined from one open-source project. However, the number of patterns that PAR uses is much smaller than the number of all bug fix patterns defined in the literature. For instance, Pan et al. [9] present 27 bug fix patterns. Moreover, no repair approach considers the abundance of each bug fix pattern. Pan et al.'s work measures the importance of them and shows that there are patterns that are more important (i.e., frequent) than others. Our intuition is that all this information could be useful to increase the repairability of defects. There are repairs that are more frequent than others. We aim at defining a repair strategy that focuses first on frequent repair done by developers.

Secondly, we focus on the limitations of the previous works that analyze the software evolution. In particular, we study those that discover change patterns, and those that measure the abundance of change patterns in the history of projects. The first limitation is they do not define a formalization of change patterns. For instance, Pan et al. [9] describe each pattern with a short paragraph and an example. As consequence, it can exist ambiguity in the pattern definition. To our knowledge, nobody has defined an approach for formalizing change patterns. In this thesis we aim at defining a method to specify change pattern. Then, using this formalization we want to define an approach that collects instances of those formalized patterns from the software history.

Another limitation relies on tools that measure the importance of change patterns (i.e., tool that collects instances of a pattern) such as SEP from Pan et al. [9]. These tools encode the pattern definitions inside their code. That means, the tool has source code that describes

each change pattern and source code to collect instances in, for instance, version control systems. So, with new proposed patterns such as those from Nath et al. [64] the tool cannot be extended to collect instances of them without modify the source code. To our knowledge, nobody has presented an approach to collect change pattern instances described from pattern formalization.

Third, the evaluation of automatic program repair approaches sometimes includes an experiment to compare the performance of a novel approach under evaluation against other approaches from the literature. Unfortunately, these evaluations are not always well-defined. For instance, the defect dataset used in an evaluation could not have explicit inclusion criteria, and this could produce a risk of biased result. We aim at defining a method to define datasets for test suite-based repair approaches evaluation with a minimized amount of bias in their definition.

In the remaining of this thesis we aim at contributing to remove the mentioned limitations presented in related work.

Chapter 3

Learning from Human Repairs by Mining Source Code Repositories

Software developers deal with bugs day to day during the development and maintenance phases. To repair a bug, a developer usually follows these steps. First, she tries to reproduce the error. Then, she finds the root cause, the bug. She creates a candidate repair and verifies whether the candidate repair fixes the bug. Finally, the developer makes visible the repair i.e., the patch is integrated into the program. For instance, the repair is committed to a version control system, which registers every change done over the program. Concurrently, automatic software repair approaches usually follow similar steps. They first localize candidate bug location, then synthesize candidate repairs, and finally validate the repairs.

In our work, we wonder whether information from human repairs could be used in automatic repair approaches. For instance, repair approaches that consider the knowledge about what are the source code changes done to fix bug and how often these changes occur. Our hypothesis is the mentioned information can be used in automatic repair approaches to increase the repairability of defects.

In this chapter we aim at extracting knowledge from bug fixes done by developers. We have two main motivations. The first one is to know what kinds of source code changes developers apply to repair bugs. For example, a repair could be to change a value to a variable, another could be to add a missing method invocation. We aim at defining models that include the source code changes used by developers for bug fixing. We study repairs done by developers at two different granularity levels. First, we study the repairs analyzing the changes at the *AST level*. For example, we study the percentage of repairs that include, at least, one change in assignment statements. Then, we study repairs at the *pattern* level. That means, we study the structure of the repair, i.e., how changes are frequently combined to fix a bug.

Our second motivation is to know which changes are more frequent when developers fix bugs. Our challenge is to measure the abundance of each kind of bug fix. In this way, automatic software repair approaches could concentrate on those repairs that are frequently done by developers.

In this chapter we study *version control systems*. These systems store source code changes made by developers during the software lifecycle. Developers commit their changes to version control systems through *versioning transactions* (also known as *commits*). A versioning

transaction introduces one or more changes done in the program under development or maintenance. We aim at describing those versioning transactions. For instance, we want to know what are the more frequent changes they introduce. Then, we aim at focusing on those transactions that introduce bug fixes. The experiments of this chapter allows us to better understand how software evolves and to characterize the bug fixing activity.

The chapter continues as follows. Section 3.1 presents two change models to describe versioning transaction. Section 3.2 presents techniques to filter those versioning transactions that contain bug fixes. Section 3.3 presents two probabilistic repair models from bug fix transactions. Section 3.4 presents a repair model formed by bug fix patterns.

The content of this chapter is published in the proceedings of one conference [90] and in one journal [11].

3.1 A Novel Way to Describe Versioning Transactions using Change Models

Software versioning repositories (managed by version control systems such as CVS, SVN or Git) store the source code changes made by developers during the software lifecycle. Version control systems (VCS) enable developers to query versioning transactions based on revision number, authorship, etc. For a given transaction, VCS can produce a difference ("diff") view that is a line-based difference view of source code. For instance, let us consider the diff presented in listing 3.1.

Listing 3.1: Example of line-based difference

```
1   while(i < MAX_VALUE){
2     op.createPanel(i);
3 -   i=i+1;
4 +   i=i+2;
5   }
```

The difference shows one line replaced by another one: line 4 per line 3. We observe that this representation does not provide much information about the change. It describes the change at a coarser granularity: a line. We only know that a statement (a line) have been updated, i.e. one statement was removed (line 3), and another was inserted in the same place (line 4). Even for this trivial source code example, it is hard to realize that it is the change about. One has to read the removed statement, then the added, and finally compare them.

However, one could also observe the changes at the abstract syntax tree (AST) level, rather than at the line level. In this case, the AST diff is an update of an assignment statement within a *while* loop. The AST diff gives us two main advantages. First, it works at a finer granularity. It focuses only in those source code elements (AST nodes) that have changed. Then, it identifies the affected elements by the change. For instance, the change could affect an assignment or a loop condition. At this level, humans are able to easier understand the changes.

Our goal in this section is study the content of versioning transactions to know how developers evolve a given program. For that, we need a mechanism to describe versioning transactions. Previous empirical studies on versioning transactions [41, 40, 39, 38, 55] focus on metadata (e.g., authorship, commit text) or size metrics (number of changed files, number of hunks, etc.). However, we aim at describing versioning transactions in terms of con-

tents: what kind of source code change they contain: addition of method calls; modification of conditional statements; etc. We choose an AST level granularity to describe versioning transactions. We believe this level of granularity is robust enough for human (developers) to understand how software evolves. There is previous work on the evolution of source code (e.g. [25, 29, 91]). However, to our knowledge, they are all at a coarser granularity compared to what we use in this work.

Formally, the research question of this section is: *what are versioning transactions made of at the abstract syntax tree level?*

To answer our research question, we follow the following methodology. First, we choose an AST differencing algorithm from the literature. Then, we constitute a dataset of software repositories to run the AST differencing algorithm on a large number of transactions. Finally, we compute descriptive statistics on those AST-based differences.

Note that other terms exist for referring to versioning transactions: "commits", "changesets", "revisions". Those terms reflect the competition between versioning tools (e.g. Git uses "changeset" while SVN "revision") and the difference between technical documentation and academic publications which often use "transaction". In this section, we equate those terms and generally use the term "transaction", as previous research does.

The rest of the section is organized as follows. In section 3.1.1 we present an AST differencing algorithms from the literature. Then, in section 3.1.2 we present a model to describe source code changes. Finally, in section 3.1.3 we present a study that describes versioning transactions of open-source projects using the defined change model.

3.1.1 Abstract Syntax Tree Differencing

There are different propositions of AST differencing algorithms in the literature. Important ones include Raghavan et al.'s Dex [18], Neamtiu et al's AST matcher [19] and Fluri et al's ChangeDistiller [20]. For our empirical study on the contents of versioning transactions, we have selected the latter.

ChangeDistiller [20] is a fine-grain AST differencing tool for Java. We list the most important reason, in our opinion, for selecting this algorithm. First, it expresses fine granularity source code changes using a taxonomy of 41 source changes types, such as "statement insertion" of "if conditional change". ChangeDistiller handles changes that are specific to object-oriented elements such as "field addition", "method declaration". The smallest element used are statements. Then, Fluri and colleagues have published an open-source stable and reusable implementation of their algorithm for analyzing AST changes of Java code. We use this implementation to develop our experiments.

ChangeDistiller produces a set of "source code changes" for each pair of Java files from versioning transactions. For a source code change, the main output of ChangeDistiller is a "change type" (from the taxonomy aforementioned). However, for our analysis, we also consider two other pieces of information. We describe the output of ChangeDistiller as follows. Each AST source code change is represented as a 2-value tuple: $scc = (ct, et)$ where *ct* is one of the 41 change types, *et* (for entity type) refers to the source code entity related to the change (for instance, a statement update may change a method call or an assignment). For example, the listing 3.1 would be represented as one single AST change that is a statement update (ct) of an assignment (*et*). Since ChangeDistiller is an AST differencer, formatting transactions (such as changing the indentation) produce no AST-level change at all.

3.1.2 Definition of Change Models

All versioning transactions can be expressed within a "change model". We define a change model as a set of "change actions". For instance, the change model of standard Unix diff is composed of two change actions: line addition and line deletion. A change model represents a kind of feature space, and observations in that space can be valued. For instance, a standard Unix diff produces two integer values: the number of added lines and the number of deleted lines. ChangeDistiller enables us to define the following change models.

CT (*Change Type*) is composed of 41 features, the 41 change types of ChangeDistiller. For instance, one of these features is "Statement Insertion" (we may use the shortened name "Stmt_Insert"). **CTET** (*Change Type Entity Type*) is made of all valid combinations of the Cartesian product between change types and entity types. CTET is a refinement of CT. Each change action of CT is mapped to $[1 \dots n]$ change actions of CTET. Hence the labels of the change actions of CTET always contain the label of CT. There are 104 entity types and 41 change types but many combinations are impossible by construction, as a result CTET contains 173 features. For instance, since there is one entity type representing assignments, one feature of CTET is "statement insertion of an assignment".

3.1.2.1 Presenting Probabilistic Change Models

A *probabilistic change model* is a change model where each "change action" has associated a value with the probability that this action occurs.

We define two measures for a change action i: α_i is the absolute number of change action i in a dataset; χ_i is the probability of observing a change action i as given by its frequency over all changes ($\chi_i = \alpha_i / \sum \alpha_i$). For instance, let us consider feature space CT and the change action "statement insertion" (StmtIns). If there is $\alpha_{StmtIns} = 12$ source code changes related to statement insertion among 100, the probability of observing a statement insertion is $\chi_{StmtIns} = 12\%$.

3.1.2.1.1 Computing Measures from Versioning Transactions Measures α and χ implicitly depend on the set of transactions that are computed. We call *transaction bag* to a set of transaction that contains a defined inclusion criterion. For example, one can define a transaction bag of all transactions done by one developer or another that includes all transactions that add one method invocation. In this section, we consider that all transactions from the version control system are included in the transaction bug. Further, in Section 3.3 we define another kind of transaction bug: *bug fix transaction bugs*.

In the rest of this section, we express versioning transactions within CT and CTET change models. There is no better change model per se: they describe versioning transactions at different granularity. In Section 4.1.3.2.2 we show that, depending on the perspective, both change models have pros and cons.

3.1.3 Empirical Evaluation

In this section, we aim at responding our research question: *What are versioning transactions made of at the abstract syntax tree level?*

To respond this question, we present a study about the content of versioning transactions of 14 repositories of Java software. We first describe versioning transactions using change

Change Action	α_i	Prob. χ_i
Statement insert	345,548	28.9
Statement delete	276,643	23.1
Statement update	177,063	14.8
Statement parent change	69,425	5.8
Statement ordering change	56,953	4.8
Additional functionality	49,192	4.1
Condition expression change	42,702	3.6
Additional object state	29,328	2.5
Removed functionality	26,172	2.2
Alternative part insert	20,227	1.7
Total	**1,196,385**	

Table 3.1: The Top-10 AST-level Changes of Change Model CT Represented Among 62.179 Versioning Transactions.

models CT and CTET. Then, we calculate the probability distribution of the change actions for those models.

3.1.3.1 Dataset

CVS-Vintage is a dataset of 14 repositories from open-source Java software [92]. The inclusion criterion of CVS-Vintage is that the repository mostly contains Java code and has been used in previous published academic work on mining software repositories and software evolution. This dataset covers different domains: desktop applications, server applications, libraries such as logging, compilation, etc. It includes the repositories of the following projects: ArgoUML, Columba, JBoss, JHotdraw, Log4j, org.eclipse.ui.workbench, Struts, Carol, Dnsjava, Jedit, Junit, org.eclipse.jdt.core, Scarab and Tomcat. In all, the dataset contains 89,993 versioning transactions, 62,179 of them have at least one modified Java file. Overtime, 259,264 Java files have been revised (which makes a mean number of 4.2 Java files modified per transaction).

3.1.3.2 Empirical Results

We have run ChangeDistiller over the 62,179 Java transactions of our dataset, resulting in 1,196,385 AST-level changes for both change models. Table 3.1 presents the top 10 change actions and the associated measures for change model CT. For change model CT, which is rather coarse-granularity, the three most common changes are "statement insert" (28% of all changes), "statement delete" (23% of all changes) and "statement update" (14% of all changes). Some changes are rare, for instance, "addition of class derivability" (adding keyword `final` to the class declaration) only appears 99 times (0.0008% of all changes).

Table 3.2 presents the top 20 change actions and the associated measures for change model CTET. One sees that inserting method invocations as statement is the most common change, which makes sense for open-source object-oriented software that is growing.

Let us now compare the results over change models CT and CTET. One can see that statement insertion is mostly composed of inserting a method invocation (6.9%), insert an "if" statement (6.6%), and insert a new variable (4.6%). Since change model CTET is at a finer granularity, there are fewer observations: both α_i and χ_i are lower. The probability distribution (χ_i) over the change model is less sharp (smaller values) since the feature space is bigger. High value of χ_i means that we have a change action that can frequently be found

Change Action	α_i	Prob. χ_i
Statement insert of method invocation	83,046	6.9%
Statement insert of if statement	79,166	6.6%
Statement update of method invocation	76,023	6.4%
Statement delete of method invocation	65,357	5.5%
Statement delete of if statement	59,336	5%
Statement insert of variable declaration statement	54,951	4.6%
Statement insert of assignment	49,222	4.1%
Additional functionality of method	49,192	4.1%
Statement delete of variable declaration statement	44,519	3.7%
Statement update of variable declaration statement	41,838	3.5%
Statement delete of assignment	41,281	3.5%
Condition expression change of if statement	40,415	3.4%
Statement update of assignment	34,802	2.9%
Addition of attribute	29,328	2.5%
Removal of method	26,172	2.2%
Statement insert of return statement	24,184	2%
Statement parent change of method invocation	21,010	1.8%
Statement delete of return statement	20,880	1.7%
Insert of else statement	20,227	1.7%
Deletion of else statement	17,197	1.4%
Total	**1,196,385**	

Table 3.2: The abundance of AST-level changes of change model CTET over 62,179 versioning Transactions. The probability χ_i is the relative frequency over all changes (e.g. 6.9% of source code changes are insertions of method invocation).

in real data: those change actions have of a high coverage of data. CTET features describe modifications of software at a finer granularity. *The differences between those two change models illustrate the tension between a high coverage and the analysis granularity.* For example, let us suppose an algorithm that predicts the changes that a versioning transaction contains. If we describe a transaction using the CT model, the prediction algorithm would have more probability to predict correctly changes than in the case the transaction is described using the CTET model. Remember CT model has fewer elements than CTET. However, after a correct prediction, the algorithm has *a more complete picture* of the transaction, i.e., more detailed information, when CTET model is used. The main reason of that is the CTET model is *more descriptive* than the CT model. Its elements have more information within, for instance, the type of entity affected by the change. In Section 4.1.3.2.2 we show that the tension between those models exists in the process of synthesizing bug fixes.

3.1.3.3 Project-independence of Change Models

An important question is whether the probability distribution (composed of all χ_i) of Tables 3.1 and 3.2 is generalizable to Java software or not. That is, do developers evolve software in a similar manner over different projects? To answer this question, we have computed the metric values not for the whole dataset, but per project. In other words, we have computed the frequency of change actions in 14 software repositories. We would like to see that the values do not vary between projects, which would mean that the probability distributions over change actions are project-independent. Since our dataset covers many different domains, having high correlation values would be a strong point towards generalization.

Correlation is a statistical measure of the strength of a linear relationship between paired data. It is used to measure the dependence between two variables. We compute the correlation values between the probability distributions of all pairs of project of our datasets (i.e. $\frac{14*13}{2} = 91$ combinations). One correlation value takes as input two vectors representing the probability distributions (of size 41 for change model CT and 173 for change model CTET).

As correlation metric, we use Spearman's ρ [93]. We choose Spearman's ρ because it is non-parametric. In our case, what matters is to know whether the importance of change actions is similar. For instance, that "Statement Update" is more common than "Condition Expression Change". The importance of a change corresponds to its *ranking*, i.e., the position of the change in the list of changes ordered (in decreasing manner) by the probability χ_i. For example, "Statement Update" is the 3rd most frequent change in Table 3.1 (χ_i = 14.8), while "Condition Expression Change" is the 7th (χ_i = 3.6). Contrary to parametric correlation metric (e.g. Pearson [94]), Spearman's ρ only focuses on the ordering between change actions, which is what we are interested in.

Spearman's correlation coefficient ρ measures the strength of association between two ranked variables. The closer ρ is to ± 1 the stronger the relationship between the two variables. The closer ρ is to 0, the weaker the association between the ranks. For instance, a ρ = 1 indicates a perfect association of ranks, a ρ = 0 indicates no association between ranks and a ρ = -1 indicates a perfect negative association of ranks. The critical value of Spearman's ρ depends on size of the vectors being compared and on the required confidence level. At confidence level $\alpha = 0.01$, the critical value for change model CT with 41 features is 0.364 and is 0.301[8] for change model CTET (values from statistical tables, we used [95]). If the correlation is higher than the critical value, the null hypothesis (a random distribution) is rejected.

For instance, in change model CT, the Spearman's correlation between Columba and ArgoUML is 0.94 which is much higher than the critical value (0.364). This means that the correlation is statistically significant at $\alpha = 0.01$ confidence level. The high value shows that both projects were evolved in a very similar manner. All values are given in A. Figure 3.1 gives the distribution of Spearman's correlation values for change model CT. 75% of the pairs of projects have a Spearman's correlation higher than 0.85[9]. For all project pairs, in change model CT, Spearman's ρ is much higher that the critical value. *This shows that the likelihood of observing a change action is globally independent of the project used for computing it.*

To understand the meaning of those correlation values, let us now analyze in detail the lowest and highest correlation values. The highest correlation value is 0.98 and it corresponds to the project pair Eclipse-Workbench and Log4j. In this case, 33 out of 41 change actions have a rank difference between 0 and 3. The lowest correlation value is 0.80 and it corresponds to Spearman's correlation values between projects Tomcat and Carol. In this case, the maximum rank change is 23 (for change action "Removing Method Overridability" — removing *final* for methods). In total, between Tomcat and Carol, there are six change actions for which the importance changes of at least 10 ranks. Those high values trigger the 0.80 Spearman's correlation. However, for common changes, it turns out that their ranks do not change at all (e.g. for "Statement Insert", "Statement Update", etc.).

We have also computed the correlation between projects within change model CTET (see

[8]Most statistical tables of Spearman's ρ stop at N=60, however since the critical values decrease with N, if $\rho > 0.301$ the null hypothesis is still rejected.

[9]Spearman's correlation is based on ranks, a value of 0.85 means either that most change actions are ranked similarly or that a single change action has a really different rank.

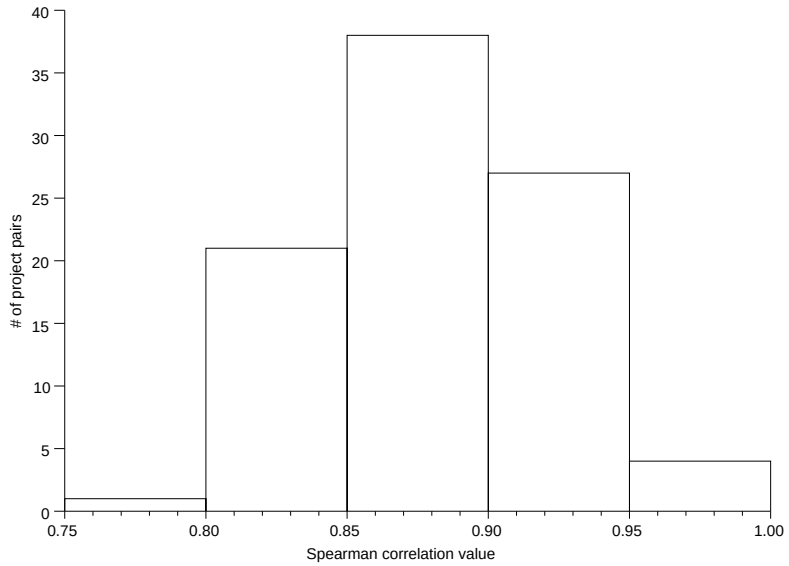

Figure 3.1: Histogram of the Spearman correlation between changes action frequencies of change model CT mined on different projects. There is no outlier, there are all higher than 0.75, meaning that the importance of change actions is project-independent.

A). They are all above 0.301, the critical value for vectors of size 173 at $\alpha = 0.01$ confidence level, showing that in change model CTET, the change action importance is project-independent as well, in a statistically significant manner. Despite being high, we note that they are slightly lower than for change model CT, this is due to the fact that Spearman's ρ generally decreases with the vector size (as shown by the statistical table).

3.1.3.4 Threats to Validity

The threats to the validity of our results are of two kinds. From the internal validity viewpoint, a bug somewhere in the implementation or in the third-party tools used (AST diff) may invalidate our results. From the external validity viewpoint, there is risk that our dataset of 14 projects is not representative of Java software as a whole, even if they are written by different persons from different organizations in different application domains. Also, our results may not generalize to other programming languages.

3.1.3.5 Summary

In this section we learned how to build change models that represent changes done by developers during the software evolution. These models allow us to understand how software evolves i.e., what are the source code changes that are done to evolve software and their frequencies. In particular, in this section we provided the empirical importance of 173 source code change actions; we showed that the importance of change actions is project independent; we showed that the probability distribution of change actions is very unbalanced. Our

results are based on the analysis of 62,179 transactions. In the remaining sections of this chapter we focus on representing models built from those changes that fix bugs.

3.2 Techniques to Filter Bug Fix Transactions

In Section 3.1.2 we have defined two probabilistic change models CT and CTET. Both models describe all types of source code changes that occur during software evolution. For each of their change action, we calculated two measures, α and χ, considering all versioning transactions of the repository.

In this section we focus on *bug fix transactions*. A bug fix transaction contains changes to fix a bug. Popular version control systems such as CVS, GIT or SVN do not provide a mechanism to label a transaction that introduces a fix as a bug fix transaction. The challenge of this section is to define criteria to filter those bug fix transactions. The transactions that fulfill a criterion are grouped in a *transaction bag*, defined in section 3.1.2.1. In this section we first present two criteria. Then, we measure the metrics α_i and χ_i (see Section 3.1.2.1) for each transaction bug. Before going further, let us clarify the goal of the transaction classification: the goal is to have a good approximation of the probability distribution of change actions for software repair[10].

The rest of the section is organized as follows. In Section 3.2.1 we present a criterion to filter transactions based on commit messages. In Section 3.2.2 we present a second filtering criterion based on the number of AST changes associated to a versioning transaction. In Section 3.2.3 we present a study to validate that transactions with a small number of AST changes are related to bug fixing activity.

3.2.1 Slicing Based on the Commit Message

When committing source code changes, developers may write a comment explaining the changes they have made. For instance when a transaction is related to a bug fix, they may write a comment referencing the bug report or describing the fix.

To identify transaction bags related to a bug fix, previous work focused on the content of the commit text: whether it contains a bug identifier, or whether it contains some keywords such as “fix” (see [50] for a discussion on those approaches). To identify bug fix patterns, Pan et al. [9] select transactions containing at least one occurrence of “bug”, “fix” or “patch”. We call this transaction bag BFP. The acronym BFP comes from the words Bug, Fix and Patch. For example, in log4j project the developer C.Gulcu introduced a fix in the revision 310908 and wrote in its commit message log: “Fixed an infinite loop bug in the AppenderSkeleton guard logic.” This transaction fulfills the BFP criterion and can be included in the BFP transaction bag.

Such a transaction bag makes a strong assumption on the development process and the developer’s behavior: it assumes that developers generally put syntactic features in commit texts enabling to recognize repair transactions, which is not really true in practice [50, 49, 47]. For instance, there are transactions that include the word "bug" or "fix" but indeed they do not introduce bug fixing source code.

[10]Note that our goal is not to have a good classification in terms of precision or recall.

3.2.2 Slicing Based on the Change Size in Terms of Number of AST Changes

We may also define fixing transaction bags based on their "AST diffs", i.e., based on the type and numbers of change actions that a versioning transaction contains. This transaction bag is called N-SC (for N Abstract Syntactic Changes), e.g. 5-SC represents the bag of transactions containing five AST-level source code changes.

In particular, we assume that small transactions are likely to only contain a bug fix and unlikely to contain a new feature. Change actions may be those that appear atomically in transactions (i.e., the transaction only contains one AST-level source code change). "1-SC" (composed of all transactions of one single AST change) is the transaction bag that embodies this assumption. Let us verify this assumption.

3.2.3 Do Small Versioning Transactions Fix Bugs?

In Section 3.2 we present a definition of transaction bag based on the type and numbers of changes that a transaction introduces. In this section we aim at determining whether small transactions correspond to bug fix changes. In particular, we define small as those transactions that introduce only one AST change.

3.2.3.1 Overview

The study consists in manual inspection and evaluation of source code changes of versioning transactions. First, we randomly take a sample set of transactions from our dataset (see 3.1.3.1). Then, we create an *evaluation item* for each pair of files from the sample set (the file before and after the revision). A *rater* is a person who decides whether an evaluation item corresponding to a bug fix or not. An evaluation item contains enough data to help the raters to carry out that decision. To help understanding the changes, it includes the syntactic line-based differencing between the revision pair of the transaction. Moreover, it includes information about the changes between at AST level such as the change type and location, e.g., insertion of method invocation at line 42. Finally, evaluation item shows the commit message associated with the transaction.

3.2.3.2 Sampling Versioning Transactions

We use stratified sampling to randomly select 1-SC versioning transactions from the software history of 16 open source projects (mostly from [92]). Recall that a "1-SC" versioning transaction only introduces one AST change. The stratification consists of picking 10 items (if 10 are found) per project. In total, the sample set contains 144 transactions sampled over 6,953 1-SC transactions present in our dataset.

3.2.3.3 Evaluation Procedure

The 144 evaluation items were evaluated by three raters: the author of this thesis and two University professors. During the evaluation, each item (see 3.2.3.1) is presented to a rater, one by one. The rater has to answer the question *Is a bug fix change?*. The possible answers are *a*) *Yes, the change is a bug fix*, *b*) *No, the change is not a bug fix* and *c*) *I don't know*. Optionally, the rater can write a comment to explain his decision.

	Full Agreement (3/3)	Majority (2/3)
Transaction is a Bug Fix	74	21
Transaction is not a Bug Fix	22	23
I don't know	0	1

Table 3.3: The Results of the Manual Inspection of 144 Transactions by Three Raters.

3.2.3.4 Experiment Results

3.2.3.4.1 Level of Agreement The three raters fully agreed that 74 of 144 (51.8%) transactions from the sample transactions are bug fixes. If we consider the majority (at least 2/3 agree), 95 of 144 transactions (66%) were considered as bug fix transactions. The complete rating data is given in A.

Table 3.3 presents the number of agreements. The column *Full Agreement* shows the number of transactions for which all raters agreed. For example, the three rates agreed that there is a bug fix in 74/144 transactions. The *Majority* column shows the number of transactions for which two out of three raters agree. To sum up, small transactions predominantly consists of bug fixes.

Among the transactions with full agreement on the absence of bug fix changes, the most common case found was the addition of a method. This change indeed consists of the addition of one single AST change (the addition of a "method" node). Interestingly, in some cases, adding a method was indeed a bug fix, when polymorphism is used: the new method fixes the bug by replacing the super implementation.

3.2.3.4.2 Statistics Let us assume that p_i measures the degree of agreement for a single item. In this experiment is $\{\frac{1}{3}, \frac{2}{3}, \frac{3}{3}\}$. The overall agreement $\bar{P}$ [96] is the average over the degree of agreement p_i. We have $\bar{P} = 0.77$. Using the scale introduced by [97], this value means there is a *Substantial* overall agreement between the rates, close to an *Almost perfect agreement*.

The coefficient κ (Kappa) measures the confidence in the agreement level by removing the chance factor[11] [96, 98]. The κ degree of agreement in our study is 0.517, a value distant from the critical value which is 0. The null hypothesis is rejected, the observed agreement was not due to chance.

3.2.3.5 Conclusion

The manual inspection of 144 versioning transactions shows that there is a relation between the one AST change transactions and bug fixing. By consequence, we can use the 1-SC transaction bag to estimate the probability of change actions for software repair.

3.3 Learning Repair Models from Bug Fix Transactions

As discussed in Section 3.1 , a change model describes all types of source code change that occur during software evolution. Now, we aim at defining a change model made from a

[11] Some degree of agreement is expected when the ratings are purely random[96, 98].

subset of the software evolution: the bug fixing. We call "repair model" to this kind of models. This section presents how we transform a "change model" into a "repair model" usable for automated software repair.

We define a "repair action" as a change action that often occurs for repairing software, i.e. often used for fixing bugs. By construction, we define a repair model as a subset of a change model in terms of features. But more than the number of features, our intuition is that the probability distribution over the feature space would vary between change models and repair models. For instance, one might expect that changing the initialization of a variable has a higher probability in a repair model. Hence, the difference between a change model and a repair model is matter of perspective. Since we are interested in automated program repair, we now concentrate on the "repair" perspective hence use the terms "repair model" and "repair action" in the rest of this chapter.

In this section we define repair models from the transaction bags presented in 3.2. The result of this section shows that, depending on the transaction bug criteria used, we obtain different topologies for repair models.

ALL	BFP	1-SC	5-SC	10-SC	20-SC
Stmt_Insert-29%	Stmt_Insert-32%	Stmt_Upd-38%	Stmt_Insert-28%	Stmt_Insert-31%	Stmt_Insert-33%
Stmt_Del-23%	Stmt_Del-23%	Add_Funct-14%	Stmt_Upd-24%	Stmt_Upd-19%	Stmt_Del-16%
Stmt_Upd-15%	Stmt_Upd-12%	Cond_Change-13%	Stmt_Del-11%	Stmt_Del-14%	Stmt_Upd-16%
Param_Change-6%	Param_Change-7%	Stmt_Insert-12%	Add_Funct-10%	Add_Funct-8%	Param_Change-7%
Order_Change-5%	Order_Change-6%	Stmt_Del-6%	Cond_Change-7%	Param_Change-7%	Add_Funct-7%
Add_Funct-4%	Add_Funct-4%	Rem_Funct-5%	Param_Change-5%	Cond_Change-6%	Cond_Change-5%
Cond_Change-4%	Cond_Change-3%	Add_Obj_St-3%	Add_Obj_St-3%	Add_Obj_St-3%	Add_Obj_St-3%
Add_Obj_St-2%	Add_Obj_St-2%	Order_Change-2%	Rem_Funct-3%	Rem_Funct-2%	Order_Change-3%
Rem_Funct-2%	Alt_Part_Insert-2%	Rem_Obj_St-2%	Order_Change-1%	Order_Change-2%	Rem_Funct-2%
Alt_Part_Insert-2%	Rem_Funct-2%	Inc_Access_Change 1%	Rem_Obj_St-1%	Alt_Part_Insert-1%	Alt_Part_Insert-2%
C1	C2	C3	C4	C5	C6

Table 3.4: Top 10 Change Types of Change Model CT and their Probability χ_i for Different Transaction Bags. The different heuristics used to compute the fix transactions bags has a significant impact on both the ranking and the probabilities.

3.3.1 Methodology

We have applied the same methodology as in 3.1.2. We have computed the probability distributions of repair model CT and CTET based on different definitions of fix transactions, i.e. we have computed α_i and χ_i based on the transactions bags discussed in 3.2: ALL transactions (column 6 in Table 3.4), BFP (column 2), and N-SC. For N-SC, we choose four values of N: 1-SC, 5-SC, 10-SC and 20-SC (columns 3, 4, 5 and 6, respectively). Transactions larger than 20-SC have almost the same topology of changes as ALL, as we will show later (see section 3.3.3.2).

The research question we ask in this section is: *Do different definitions of "repair transactions" (ALL, BFP, N-SC) yield different topologies for repair models?*

3.3.2 Empirical Results

Table 3.4 presents the top 10 change types of repair model CT associated with their probability χ_i for different versioning transaction bags. Overall, the distribution of repair actions over real bug fix data is very unbalanced, the probability of observing a single repair action goes from more than 30% to 0.000x%. We observe the Pareto effect: the top 10 repair actions account for more than 92% of the cumulative probability distribution.

Furthermore, we have made the following observations from the experiment results. First, the order of repair actions (i.e. their likelihood of contributing to bug repair) varies significantly depending on the transaction bag used for computing the probability distribution. For instance, Table 3.4 shows that a statement insertion is #1 when we consider all transactions (column ALL), but only #4 when considering transactions with a single AST change (column 1-SC). In this case, the probability of observing a statement insertion varies from 29% to 12%.

Second, even when the orders obtained from two different transaction bags resemble such as for ALL and 20-SC, the probability distribution still varies: for instance χ_{Stmt_Upd} is 29% for transaction bag ALL, but jumps to 33% for transaction bag 20-SC.

Third, the probability distributions for transaction bags ALL and BFP are close: repair action has similar probability values. As consequence, transaction bag BFP maybe is a random subset of ALL transactions. All those observations also hold for repair model CTET, the complete table is given in the appendix A. Those results are a first answer to our question: *different definitions of "bug fix transactions" yield different probability distributions over a repair model.* That means, there are changes that occur more frequent in a particular kind of transactions than in others. It could have an immediate implementation: by considering this information, repair approaches could be able to focus first in those frequent repair actions.

3.3.3 Discussion

We have shown that one can base repair models on different methods to extract repair transaction bags. There are certain analytical arguments against or for those different repair space topologies. For instance, selecting transactions based on the commit text makes a very strong assumption on the quality of software repository data, but ensures that the selected transactions contain at least one actual repair. Alternatively, small transactions indicate that they focus on a single concern, they are likely to be a repair. However, small transactions may only see the tip of the fix iceberg (large transactions may be bug fixing as well), resulting in a distorted probability distribution over the repair space. At the experimental level, the threats to validity are the same as for Section 3.1.2.

3.3.3.1 Correlation between Transaction Bags

	1-SC	5-SC	10-SC	20-SC	BFP
ALL	0.68	0.95	0.97	0.98	0.99

Table 3.5: The Spearman correlation values between repair actions of transaction bag "ALL" and those from the transaction bags built with 5 different heuristics.

In this section we present a study to know to what extent the 6 transactions bags are different. We have calculated the Spearman correlation values between the probabilities over repairs actions between all pairs of distribution. In particular, we would like to know whether the heuristics yield significantly different results compared to all transactions (transaction bag ALL). Table 3.5 presents these correlation values.

For instance, the Spearman correlation value between ALL and 1-SC is 0.68. This value shows, as we have noted before, that there is not a strong correlation between the order of their repair actions of both transaction bags. In other words, heuristic 1-SC indeed focuses on a specific kind of transactions.

On the contrary, the value between ALL and BFP is 0.99. This means the order for the frequency of repair actions are almost identical. Moreover, Table 3.5 shows the correlation values between N-SC (N = 1, 5, 10 and 20) and ALL tend to 1 (i.e., perfect alignment) when N grows. This validates the intuition that the size of transactions (in number of AST changes) is a good predictor to focus on transactions that are different in nature from the normal software evolution. Crossing this result with the results of our empirical study of 144 1-SC transactions (see Section 3.2.3), there is some evidence that by concentrating on small transactions, we get a good approximation of repair transactions.

3.3.3.2 Skewness of Probability Distributions

Figure 3.2 shows the probability for the most frequent repair actions of repair model CTET according to the transaction size (in number of AST changes). For instance, the probability of updating a method invocation decreases from 15% in 1-SC transactions to 7% in all transactions. In particular, we observe that: *a*) For transaction with 1 AST change, the change probabilities are more unbalanced (i.e. less uniform than for all transactions). There are 5 changes that are much more frequent than the rest. They are: “statement update of method invocation", “add method", “if condition change", “statement update of variable declaration", and “statement update of method invocation". *b*) For transactions with more than 10 AST changes, the probabilities of top changes are less dispersed and all smaller than 0.9% *c*) The probabilities of those 5 most frequent changes decrease when the transaction size grows. This is a further piece of evidence that heuristics N-SC provide a focus on transactions that are of specific nature, different from the bulk of software evolution.

3.3.3.3 Summary

Those results on repair actions are especially important for automated software repair: we think it would be fruitful to devise automated repair approaches that “imitate” how human developers fix programs. In Section 4.1, we use the presented repair models for reasoning on the repair search space. To us, using the probabilistic repair models as described in this section is a first step in that direction. In the next Section 3.4, we analyze bug fix transactions in another granularity: at *pattern* level. We study how the elements from the CTET models frequently appear together in bug fix transactions. From this study, we are able to define a repair model composed by bug fix patterns. This model could be used by repair approaches based on bug fix pattern such as PAR [5].

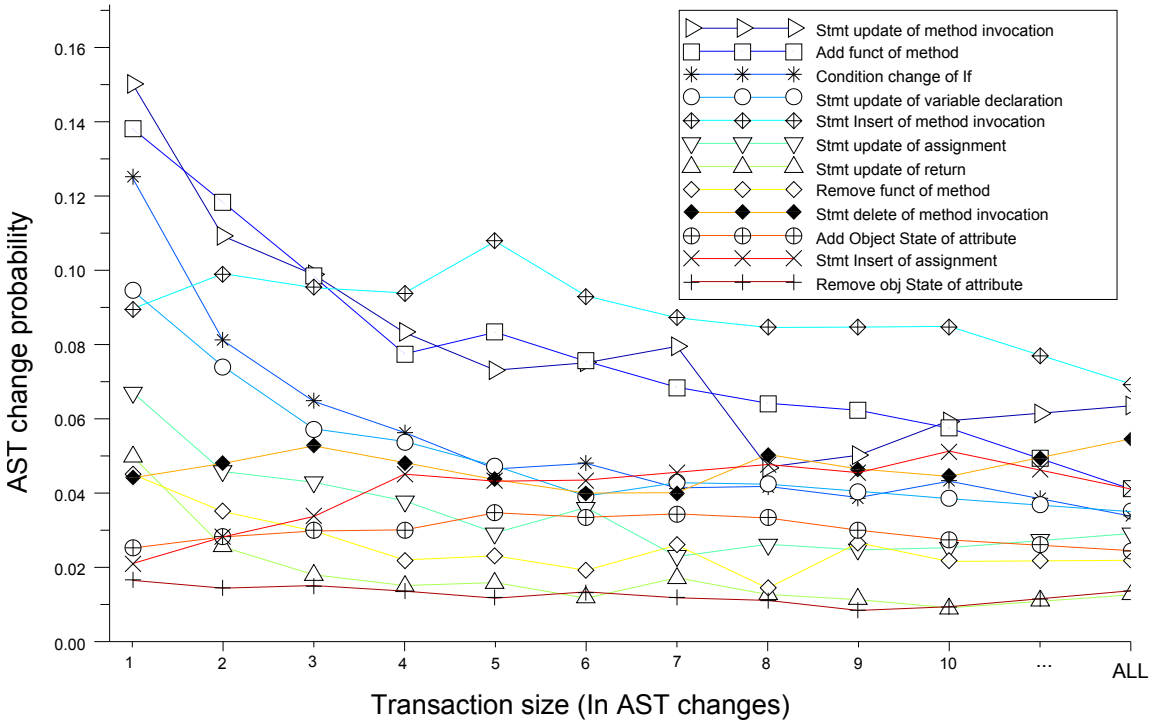

Figure 3.2: Probabilities of the 12 most frequent AST changes for 11 different transaction bags: 10 that include transactions with i AST changes, with $i = 1...10$, and the ALL transaction bag.

3.4 Defining a Repair Model of Bug Fix Patterns

In Section 3.3 we present two repair models built from bug fix transactions. Those models represent modifications of software at the AST change level. For instance, the change model CT has a change action called "statement parent change", and the CTET model has one called "statement update of assignment".

We observe changes in versioning transactions do not appear isolated. For instance, from the same corpus that experiment of Section 3.1, we observe that 3095 out of 3667 BFP transactions (84%) have two or more changes (AST changes from ChangeDistiller's granularity). That means the majority of repairs done by developers are composed of more than two of those changes.

Let us present an example. Suppose a program defines one variable a that it is not initialized. A failure occurs when the program attempts to use the variable's value. A possible fix could is presented in Listing 3.2 [12].

Listing 3.2: Example of bug fix condition

```
1 if (var a is not initialized) then
2         initialize a;
```

Let us represent using CTET repair model the transaction that introduces this fix in the version control system. The representation of this transaction has two repair actions: one for the addition of if statement, another for the initialization of the variable i.e., "addition of an assignment".

In this section we aim at presenting a repair model that describes this kind of fixes done by developers. That is, a model where each of element are a composition of one or more elements of, for instance, the CTET model. A *bug fix pattern* encodes a kind of bug fix. The model is composed of bug fix patterns defined in the literature. For example, Pan et al.[9] call *Addition of precondition check (IF-APC)* to the fix presented in the example of listing 3.2. Our challenge is to define a probabilistic repair model for automatic software repair that captures the importance of each bug fix pattern. In a probabilistic model, each of their elements contains a value related to probability that the element is selected when one aims at picking one element from the model.

The goal of this section is twofold. First, we aim at presenting a mechanism to specify bug fix patterns from the literature (Section 3.4.2). We define bug fix pattern in Section 3.4.1. Second, we aim at measuring the importance of each bug fix pattern in version control systems (Section 3.4.4. For that, we define a mechanism to search for instances of bug fix patterns. The input of this mechanism is a specification of one pattern, while the output is a set of versioning transactions that contain instances of that pattern.

We also present two experiments to evaluate those mechanisms. In the first one, we aim at measuring the genericity of the pattern specification mechanism (Section 3.4.5). This evaluation allows us to measure the capacity of the approach to encode bug fix patterns from the literature. In the second experiment, we evaluate our mechanism of search of bug fix pattern instances (Section 3.4.6). Finally, we aim at measuring the abundance of bug fix patterns (Section 3.4.7) for defining a probabilistic repair model formed by bug fix patterns.

[12] Another fix is to initialize the variable with one value when it is defined.

3.4.1 Defining Bug Fix Patterns

Bug fix patterns capture the knowledge on how to fix bugs. Bug fix patterns are essential building blocks of research areas such as automatic program repair [12, 5]. One such bug fix pattern called *Change of If Condition Expression* (IF-CC) has been identified by Pan et al. [9]. Figure 3.3 presents one instance of this pattern by showing two consecutive revisions of a source code file. Revision N (on the left-hand side) contains a bug inside the *if condition*, a wrong call to the boolean method *isEmpty* instead of a call to the method *isFull*. In revision $N + 1$ (the right-hand side piece of code) a developer fixed the bug by modifying the *if condition* expression.

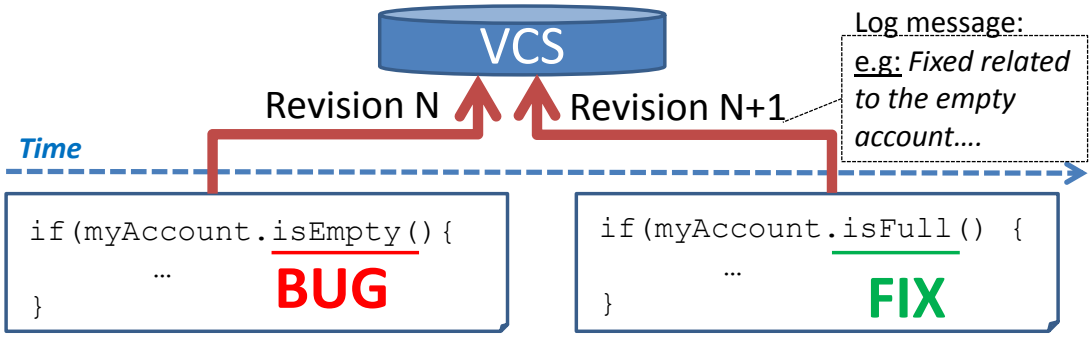

Figure 3.3: Example of a bug fix pattern called *Change of If Condition Expression (IF-CC)* [9], in two consecutive revisions of a source code file. The left-hand side revision contains a bug in the *if condition*: an incorrect method invocation. On the right-hand side, the developer fixed it by modifying the *if condition*, i.e. updating the method invocation.

3.4.2 A Novel Representation of Bug Fix Patterns based on AST changes.

Previous work such as Pan et al. [9] present catalogs of bug fix patterns. For example, Pan et al. present a catalog of 27 bug fix patterns. The authors describe each bug fix pattern with a brief textual description and one listing that shows the changes (at line level) corresponding to the pattern's instance. For example, the pattern *Change of If Condition Expression (IF-CC)* from Pan et al. is described as follows:

Description: "This bug fix change fixes the bug by changing the condition expression of an *if condition*. The previous code has a bug in the if condition logic."

Listing 3.3: Pattern *Change of If Condition Expression* defined by Pan et al.

```
- if (getView().countSelected() == 0)
+ if (getView().countSelected() <= 1)
```

Then, they measure the importance of each bug fix pattern by mining bug fix pattern instances from commits of version control systems. For that, the authors use a tool called SEP to automatically identify pattern instances. In the case of this tool, we observe a dual pattern definition. On one hand, Pan et al. present pattern definitions that target humans, such as we have seen for pattern IFCC. On the other hand, the authors encode these definitions in the same code as their tool.

In our opinion, this adoption produces some drawback. For instance, to add a new pattern it is necessary to modify the source code of the tool. Moreover, it impacts on the pattern

definition's understandability. In the case of SEP, it is necessary to inspect and debug hundreds of line of code to discover the encoded definition of one pattern.

Our motivation is to introduce a new mechanism to formalize bug fix patterns. In particular, we aim this formalization be both: *a*) human comprehensible; and *b*) used as input of mining algorithms that search pattern instances.

In this section, we present a methodology to formalize bug fix pattern from the literature. The method is based on AST analysis and tree differencing. In subsection 3.4.2.1, we present a formalization of source code changes at the AST level. Then, in subsection 3.4.2.2, we present a formalization of bug fix patterns at the AST level.

3.4.2.1 Representing Versioning Changes at the AST Level

Our method identifies bug fix pattern instances from version control system revisions. It works at the abstract syntax tree (AST) level. This means we represent a source code file revision with one AST. The advantage of this representation is it allows us to extract fine-grained changes between two consecutive revisions by applying an AST differencing algorithm. This involves representing source code revisions as changes at the AST level. As in the experiment of Section 3.1 we use ChangeDistiller as AST differencing algorithm.

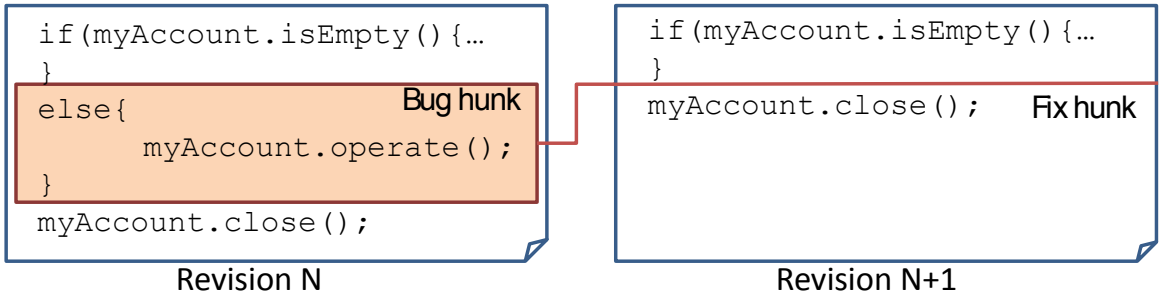

Figure 3.4: A lined-based difference of two consecutive revisions. The bug hunk in revision N (the left one) contains an "else" branch. The fix hunk in revision N+1 is empty. The corresponding AST hunk (introduced in section 3.4.3.1) consists of two nodes removal i.e. the 'else' node and the method invocation.

Let us take as example the change presented in Figure 3.4. It shows a lined-based difference (syntactic) of two consecutive revisions. The bug hunk in revision N (the left one) contains an "else" branch. The fix hunk in revision N+1 is empty. The change consists of a removal of code: removal of "else" branch. At the AST level, the AST differencing algorithm finds two AST changes: one representing the removal of an *else* node and another for the removal of a method invocation (i.e. myAccount.operate()) node surrounded by the *else* block.

As we have shown in Section 3.1, ChangeDistiller handles a set of 41 source change types included in an object-oriented change taxonomy defined by Fluri and Gall [22]. For example, the taxonomy includes source code change types such as "Statement Insertion" or "Condition Expression Change". A code change type affects object-oriented elements such as "field addition". These elements are represented by 142 entity types.

Formally, ChangeDistiller produces a list of "AST source code changes". For the pattern formalization, we need a more robust definition of AST change compared to that one used

for defining the CT and CTET models (see Section 3.1.1). For instance, we need that a change includes the location where it is done (we call it *parent entity*). We formalize each change (scc) in a 7-value tuple:

$$scc = (ct, et, id\_e, pt, id\_p, opt, id\_op)$$

where *ct* is one of the 41 change types, *et* (for entity type) refers to the source code entity type related to the change. For example, a statement update may change a method call or an assignment. The field *id_e* is the identifier of the mentioned entity. As ChangeDistiller is an AST differencing, that field corresponds to the identifier of the AST node affected by the change. The field *pt* (for parent entity type) indicates the parent code entity type where the change takes place. For example, it corresponds to a top-level method body or to an "If" block. *id_p* is the identifier of the parent entity. For change type "Statement Parent Change", which represents source code movement, pt points to the new parent element. Moreover, *opt* and *id_op* indicate the parent entity type and the identifier for the old parent entity. Both fields specify the place the moved code was located before the change occurs, and they are omitted in tuples related to changes types different from "Statement Parent Change".

Let us present two examples of AST source code changes representation. As first example, a removal of an assignment statement located inside a "For" block is represented as: $scc1$ =("Statement delete" (ct), "Assignment" (et), node_id_23 (id_e), "For" (pt), node_id_14 (id_pt)).

As a second example, a movement of an assignment located in a method body to inside an existing "Try" block located in the same method is represented as: $scc2$ =("Statement Parent Change" (ct), "Assignment" (et), node_id_24 (id_e), "Try" (pt), node_id_15 (id_p), "Method" (opt), node_id_10 (id_op)). As this change is a movement i.e. "Statement Parent Change", its tuple includes the identifiers and type from the location the code comes from (*opt* and *id_op*, both ignored in $scc1$) and the new location as well (*pt* and *id_p*).

This structure for describing changes (scc) is more complex than the representation of changes used for CT and CTET models, presented in Section 3.1.1. The structure contains more information necessary to describe pattern's changes. In particular, it includes information of the parent entity type (and eventually the new parent entity type for movement of code) to describe the entity type where the changed entity is located. Moreover, the structure includes the identifiers (*ids*) of each entity involved in the change. This allows us to link the entities (AST nodes) affected by the change. For instance, let us consider a pattern that adds an *if precondition* just before a statement. An instance of this pattern has two changes: one that corresponds to the addition of the *if*; the other a movement of the assignment (now the parent entity is the added *if* statement). The *ids* fields are used to validate whether the new location (parent entity) of the moved assignment is the added *if precondition*. In the following section we go deeper in the formalization of change patterns.

3.4.2.2 AST-based Pattern Formalization

In this section, we present a structure to formalize a bug fix pattern. This structure is used to identify bug fix pattern instances from the AST-level representation of revisions presented in section 3.4.2.1.

We specify a bug fix pattern with a structure composed by three elements: a list of micro-patterns L, a relation map R, and a list of undesired changes U.

$$\text{pattern} = \{L, R, U\}$$

In the following subsection we describe each of those pattern elements.

3.4.2.2.1 List of Micro-patterns A micro-pattern represents a change pattern over a single AST node. It is an abstraction over ChangeDistiller's AST changes, i.e., instances of source code changes of a given type. A micro-pattern is a 5-value tuple

$$mp = (ct, et, pt, opt, cardinality)$$

where *ct, et, pt* and *opt*[13] have the same meaning as the source code change formalization in section 3.4.2.1. The *ct* field is the only mandatory, while fields *et*, *pt* and *opt* can take a *wildcard* character "*", meaning they can take any value. The field *cardinality* takes a natural number that indicates the number of consecutive equivalent (with the same value in fields *ct, et* and *pt*) AST changes it represents. It also can take the value *wildcard*, meaning that the micro-pattern can represent undefined number of consecutive equivalent changes. By default (absence of explicit value in mp tuple), the cardinality is value one. For example, a micro-pattern ("Statement Insert",*,*) means that an insertion of one statement of any type (e.g., assignment) inside any kind of source code entity, e.g. "Method" (top-level method statement) or "If" block. This micro-pattern is an abstraction of all AST source code changes corresponding to the addition of one AST node, whatever the node type and place in the AST.

The list of micro-patterns L represents the changes done by the pattern. The list is ordered according to their position inside the source code file. It is not commutative: a pattern formed by micro-pattern $mp1$ followed by $mp2$ is not equivalent to another formed by $mp2$ followed by $mp1$. The former means that $mp1$ occurs before $mp2$, while the latter means the opposite.

As example, let us present the AST representation of pattern "Addition of Precondition Check with Jump" [9]. This pattern represents the addition of an *if* statement that encloses a jump statement like *return*. It is represented by two micro-patterns[14]: $mp1$ = ("Statement Insert", "If", *) and $mp2$ = ("Statement Insert", "Return", "If").

3.4.2.2.2 Relation Map The *relation map* R is a set of relations between entities involved in micro-patterns of L and U. Each relation links two entities (et, pt or opt) of two different micro-patterns. The relation r is written as:

$$r = mp1.entity_1 \text{ comp } mp2.entity_2$$

A relation formalizes a *link* between two elements (AST) from a pattern instance. That means, the elements of a pattern instance must fulfill all the relations from the pattern's relation map.

Each relation has three elements: two operands and one operator. The operator $comp$ is used to compare the related entities. In particular, we use two operators: *equal* (==) and *not equal* (!=). For example, the relation written as $mp1.pt == mp2.pt$ uses the former operator, and relation as $mp1.pt$!= $mp2.pt$ the latter.

The operands $entity_1$ and $entity_2$ specify which entity field from each micro-pattern (et, pt or opt) is involved in the relation. For instance, relation $mp1.pt == mp2.pt$ defines a

[13] We omit to specify *opt* in the tuple for addition, updates and removes operations.

[14] to simplify the example, we exclude jump statements 'break' and 'continue'.

relation between entity pt from micro-pattern $mp1$ and entity pt from micro-pattern $mp2$. This relation expresses that two changes affect entities with *the same* type of parent. Contrary, $mp1.pt$!= $mp2.pt$ expresses that two changes affect entities with a *different* type of parent.

Another case of entity relation is expressed as $mp2.pt$ == $mp1.et$. It defines that a change (matched with $mp1$) is done in an entity whose parent entity is affected by the second change (matched $mp2$).

As we mentioned, a pattern instance (i.e., a set of AST changes) must fulfill all the relations defined by the pattern. For example, let us consider a pattern P composed by micro-patterns $mp1$ and $mp2$ and one relation $R1 = (mp1.pt == mp2.et)$. Then, let us suppose that a set of changes composed by changes $scc1$ and $scc2$, are instances of $mp1$ and $mp2$, respectively. Changes $scc1$ and $scc2$ form an instance of P iff $R1$ is fulfilled by them. To verify whether those changes fulfill relation $R1$, we compare the *identifiers* of the entities affected by the changes. The first term of R1, i.e. $mp1.pt$, corresponds to the parent of $scc1$, i.e. $scc1.id\_p$. The second term of R1 ($mp2.et$) corresponds to the entity of $scc2$ ($scc.id\_e$). As consequence, the relation $R1$ is fulfilled when $scc1.id\_p$ == $scc2.id\_e$.

3.4.2.2.3 Undesired Changes Our bug fix pattern formalization is composed by a second list of micro-patterns. The list of *undesired changes* U represents micro-patterns that must not be present in the pattern instance. For example, the bug fix pattern "Removal of an Else Branch" [9] requires only the "else" branch being removed, keeping its related "if" branch in the source code. In other word, the related "if" must not be removed.

As example, let us formalize this pattern. L contains one micro-pattern $mp1$ = ("Statement delete", "else","If"), U contains one *undesired change* u_mp1= ("Statement delete", "If",*) and R contains the relation $u\_mp1.et$!= $mp1.opt$. Generally, relations associated to micro-patterns from *U* have an operator "!=" and relate a micro-pattern of *U* with another from L. Hence, the relation restricts that no undesired change be related to changes associated to micro-patterns from L. In the example, the formalization of the pattern specifies that: *a*) there is a deletion of a "else" ($mp1$); *b*) it does not exist a deletion of an "if" entity that, in turn, is the parent entity of the deleted "else" (u_mp1)).

3.4.2.2.4 Summarization In this section we have defined a structure to formalize bug fix patterns. In section 3.4.4 we present a method to identify pattern instances from this pattern formalization.

3.4.3 Defining the Importance of Bug Fix Patterns

The notion of importance of bug fix patterns refers to whether some bug fix patterns are more important than others. A measure of importance is the number of commits in which one observes an instance of the pattern, we call it the *abundance* of the pattern. The abundance reflects to what extent those bug fix patterns are used in practice.

For instance, Pan et al. report [9] that in the history of Lucene[15], the bug fix pattern "Change of if condition expression" (IF-CC) is the most common pattern with 370 instances (12% of all bug fix pattern instances identified). On the contrary, the pattern "Addition of operation in an operation sequence of field settings" (SQ-AFO) is the less abundant pattern, with only 5 instances being observed (0.2%).

[15]http://lucene.apache.org/core/

To measure the importance of one bug fix pattern we need to *identify* instances of that pattern. The *accuracy* of bug fix pattern instance identification refers to whether an approach yields the correct number of pattern instances. The threat to the accuracy of the abundance measurement of bug fix patterns is two-fold. First, one may over-estimate it by counting commits as instances of the pattern while they are actually not (false positives). Second, one may under estimate it by *not* counting commits, i.e. by missing instances (false negatives). The challenge we address is to provide a mechanism to obtain an accurate measure of bug fix pattern abundance, by minimizing both the number of false positives and the number of false negatives.

Before to present an accurate pattern instance identifier in Section 3.4.4, we present the notion of *AST* hunk.

3.4.3.1 Defining "Hunk" at the AST level

Previous work has set up the "localized change assumption" [9]. This states that the pattern instances lie in the same source file and even within a single hunk i.e., within a sequence of consecutive changed lines. For example, Figure 3.4 shows an example of two consecutive revisions of a Java file and a hunk pair representing the changes between the two revisions. The differences between the two files are grouped in consecutive changed lines which are called "hunk". In Pan et al.'s work [9], the authors identify pattern instances inside each hunk pair. As consequence, a pattern instance belongs to only one hunk pair and, by transition, to one revision file.

From our experience, the "localized change assumption" is relevant in the process of identification of bug fix pattern instances. Since we work at the level of AST and hunk are the level lines (syntactic level), we define the notion of "hunk" at the AST level. AST hunks are *co-localized source code changes*, i.e., changes that are near one from another inside the source code.

The notion of hunk is important for searching pattern instances. Our identifier aims at identifying pattern instances from AST source code changes that are in the same hunk. That means, a pattern instance never contains AST instances from different AST hunks.

For us, an AST hunk is composed of those AST changes that meet one of the following conditions: *a*) they refer to the same syntactic line-based hunk; or *b*) they are moves within the same parent element. For instance, the two AST changes from the example of Figure 3.4 are in the same AST hunk (both changes occur in the same syntactic hunk). By construction, there is no AST hunk for changes related to comments or formatting, while, at the syntactic, line based level, those hunks show up.

3.4.4 An Novel Algorithm to Identify Instances of Commit Patterns from Versioning Transactions

This section presents an algorithm to identify bug fix pattern instances inside an AST hunk (see Section 3.4.3.1). The pattern instance identifier algorithm is composed by three serial phases: *a*) change mapping (Section 3.4.4.1); *b*) exclusion of AST hunks containing undesired changes (Section 3.4.4.2); and *c*) identification of change relations (Section 3.4.4.3). Let us explain each phase in the remain of the section.

3.4.4.1 Mapping Phase

The pattern instance identification algorithm first processes the phase named *Mapping phase*. The goal of the phase is to map each micro-pattern mp_j of L (list of micro-patterns, see section 3.4.2.2) with one AST change scc_i of the hunk. The output of the phase is a map of micro-patterns and AST changes. The result of the mapping phase is successful if all micro-patterns of the bug fix pattern appear in the AST hunk i.e., they have at least one mapping with AST changes of the hunk. If this condition is not satisfied, the outcome phase is a *fail*, stopping the execution of the following phases. In other words, a pattern instance could not be identified in the hunk.

The mapping algorithm is explained in Section 3.4.4.1.2. Before, in Section 3.4.4.1.1 we detail the algorithm to match AST changes with micro patterns.

```
   Input: micro_pattern                                          ▷ Micro-pattern
   Input: change                                                 ▷ AST change (scc)
   Output: boolean value: true if the AST change change matches with the micro-pattern
           micro_pattern
1  begin
       /* First, comparison of change types                                    */
2      if micro_pattern.ct == change.ct then
           /* Then, comparison of entity types                                 */
3          if micro_pattern.et != "*" and micro_pattern.et != change.et then
4              return false;
           /* Finally, comparison of parent entity types                       */
5          if micro_pattern.pt != "*" and micro_pattern.pt != change.pt then
6              return false;
7          else
8              return true;
9      else
10         return false;
```

Figure 3.5: Algorithm to verify the matching between a micro-pattern and an AST change

3.4.4.1.1 Mapping creation criterion A change scc is mapped to the micro-pattern mp if scc is an instance of the change described by the mp. This relation is verified by matching the structures scc and mp. Algorithm 3.5 shows the matching algorithm pseudo-code. Both *match* (the matching is true) if their change types (line 2), entity types (line 3) and parent types (line 5) are the same. Note that if one wildcard (see Section 3.4.2.2) is specified, the field comparison is ignored (lines 3 and 5).

3.4.4.1.2 Mapping Algorithm Overview Let us first explain the mapping procedure and then we detail the algorithm step by step in Section 3.4.4.1.3.

First, the mapping algorithm tries to find a mapping between list of micro patterns and a sequence of AST changes of the hunk. The algorithm first searches all possible beginnings of the pattern in the hunk. A beginning is an AST change from the hunk and candidate to be the first element of the pattern inside the hunk. In other words, it must match with the first micro pattern.

Then, the algorithm tries to search a pattern instance from each of those beginnings. For each beginning, it proceeds to map the changes that follow the beginning with the list of micro-patterns. It iterates both the sequence of changes and the list of micro-patterns at the same time. A matching between a change and a micro-pattern is done in each iteration (as we explain in Section 3.4.4.1.1). If both match, the algorithm continues with the iteration, otherwise it stops analyzing the sequence and continues with the following beginning. Once the algorithm maps all micro-patterns with a sequence of changes, it returns that mapping. This sequence of AST changes is candidate to be a pattern instance.

It is important to note that the mapping phase defines two restrictions for the mappings between AST changes in a hunk and the micro-patterns. We call the first restriction *mapping total order*. It defines that the mapped AST changes must satisfy the order defined by L. Let us consider the list of micro patterns L= $\{mp1, mp2\}$ and an AST hunk H = $\{scc1, scc2\}$. The mapping $scc1$ with $mp1$ and $scc2$ with $mp2$ is valid. Let us explain why. The mapped AST changes respect the order imposed by the pattern i.e., through L, the first AST element of the hunk mapped with the first micro pattern, and so on. However, the mapping $scc1$ with $mp2$ and $scc2$ with $mp1$ is not valid. As $mp1$ appears before $mp2$ in L, then $scc2$ (mapped to $mp1$) must appear before $scc1$ in the hunk, and this is not the case. As consequence, this last mapping is not valid.

We call *consecutive mapping* to the second restrictions. It defines the mapped AST changes must be consecutive inside the hunk. In other words, it cannot exist one no-mapped change between two mapped changes. For instance, given a pattern formalization with 2 micro-patterns $mp1$ and $mp2$, and an AST hunk composed by 3 AST changes $scc1$, $scc2$ and $scc3$. Then, the mapping $mp1$, $scc1$ and $mp2$, $scc3$ is invalid due $scc2$ is not mapped and it is located between $scc1$ and $scc3$, both mapped changes.

3.4.4.1.3 Mapping Algorithm Pseudo-code Now, let us analyze the algorithm in detail. Algorithm 3.6 shows the pseudo-code of this mapping phase. The input of the algorithm is the list of micro-patterns L that represents the pattern, and a list of changes $Changes$ that represents one AST hunk.

The algorithm starts by searching a list $initial\_changes$ of AST changes. The list contains "candidates beginning" of the pattern inside the hunk i.e., in list $Changes$ (line 3). Each change of $initial\_changes$ *matches* with the first micro-pattern (see Algorithm 3.5 and explanation in Section 3.4.4.1.2).

Then, for each AST change $initial$ of the mentioned list $initial\_changes$, the algorithm tries to map all micro-patterns of L with the sequence S of consecutive AST changes that follow $initial$. The algorithm defines two cursors $change\_i$ and $micro\_pattern\_i$ to iterate the sequence S and L, respectively. In each iteration (line 6), the algorithm matches the head of both cursors (line 12) using Algorithm 3.5. If both match, the algorithm maps them and saves the association (line 13). After that, the cursors are updated (line 14 to 18 and from 22 to 23). The micro-pattern cursor is only updated once the algorithm has analyzed as many AST changes as the micro-pattern's cardinality indicates (line 16). When the cardinality is

"$*$" (wildcard), the cursor $micro\_pattern\_i$ is updated (line 22) if at least one change from S is mapped to the current micro pattern (line 20).

The algorithm finishes successfully when all micro-patterns are mapped to consecutive AST changes (line 23 and 24).

3.4.4.2 Undesired Changes Validation Phase

The second phase verifies that no change of the *undesired changes U* list is present in the hunk. The algorithm of this phase maps changes for U with AST changes from the hunk. So it is similar to that one corresponding to the *Mapping phase*.

Different from the previous phase, an empty set of mappings is a good signal: no undesired change is present in the hunk. Contrary, in case that the micro-patterns of *U* are mapped to changes of the hunk, the relations over them must be fulfilled by the phase defined in section 3.4.4.3.

3.4.4.3 Relation Validation Phase

The change relation validation phase verifies that the relations defined by the pattern's *relation map* are satisfied by the mapped AST changes of the hunk. For the validation, the maps calculated in the two previous phases (3.4.4.1 and 3.4.4.2) are used.

Algorithm 3.7 shows the corresponding pseudo-code. First, for each relation the algorithm retrieves the two micro-patterns it relates (lines 3 and 4). Then, it retrieves the AST changes mapped to those micro-patterns (lines 5 and 6). After that, the algorithm retrieves the identifiers of the entities related to the changes. For that, function $getIdFromEntityType$ first determines which kind of entities (et, pt, or opt) the relation pinpoints. Then, it returns the identifier of the corresponding entity (lines 9 and 10). Finally, the two entity identifiers are compared (line 12) according to the operator defined by the relation (line 11). The comparison involves comparing *ids* of the entities i.e., AST nodes affected by the changes. As this phase is the last one from the AST change pattern identification, a successful validation of all relations means the presence of a pattern inside the analyzed hunk.

3.4.4.4 Algorithm Result

Once all phases were executed, the pattern instance identification algorithm determines the presence of a pattern instance inside the analyzed hunk if the following conditions are valid: *a*) all micro-patterns of L are mapped and the mapped AST changes from L fulfill relations of R; and *b*) no micro-pattern of U is mapped or every mapped AST change from U fulfill relation of R.

3.4.4.5 Conclusion

In this subsection we present an algorithm to identify bug fix pattern instance in versioning transactions. This algorithm allows us to measure the importance of bug fix patterns. Then, the importance can be used in the automatic software repair field, for example, to define probabilistic repair models.

In the remain of the section we present two evaluations. In section 3.4.5, we evaluate the genericity of our bug fix pattern formalization approach. In section 3.4.6 we evaluate

the accuracy of our approach with a manual analysis of a random sample of bug fix pattern instances found in commits of an open-source project.

3.4.5 Evaluating the Genericity of the Pattern Specification Mechanism

In this section, we focus on the bug fix formalization we presented in section 3.4.2 and present an evaluation of its *genericity*. That means, we evaluate whether it is possible to specify known and meaningful bug fix patterns using our formalism.

The evaluation is built on the research question: *Can our specification format represent known and meaningful bug fix patterns?*

To answer this research question, we first present bug fix patterns from the literature in section 3.4.5.1 and a set of new bug fix patterns in section 3.4.5.2. We eventually formalize these patterns in section 3.4.5.3.

3.4.5.1 Source #1: Bug Fix Patterns from the Literature

Pan et al. [9] have defined a catalog of 27 bug fix patterns divided in 9 categories. The categories are: If-related, Method Calls, Sequence, Loop, Assignment, Switch, Try, Method Declaration and Class Field. According to the number of citations, this is one of the most important papers on bug fix patterns.

Furthermore, we also consider three additional new bug fix patterns proposed by Nath et al. [64]. The patterns are named as: "method return value changes", "scope changes" and "string literals".

The existing definitions of bug fix patterns are written in natural language and are sometimes ambiguous. Before formalizing them, we clarify four bug fix patterns from Pan et al. to facilitate their comprehension and formalization. We split into two those patterns whose definition mixes adding and removing code. For instance, "Add/removal of catch block" becomes "Add of catch block" and "Removal of catch block". Within the 27 original patterns, four of them were split, this results in a restructured catalog of 31 patterns.

3.4.5.2 Source #2: New Bug Fix Patterns

In this section, we present new meaningful bug fix patterns. We found them while browsing many commits that were done to repair bugs [11]. In section 3.4.5.3, we formalize these patterns to demonstrate the flexibility our specification mechanism.

Pattern DEC-RM: ***Deletion of variable declaration statement*** This bug fix pattern consists of the removal of a variable declaration inside the buggy method body (e.g. after a refactoring to transform a variable as field). This pattern is a sibling of Pan's patterns related to Class Field (i.e. Removal of a Class Field) but at method level.

Pattern THR-UP: ***Update of Throw Statement*** This bug fix pattern corresponds to the update of a *throw* statement. It includes changing the type of exception that is thrown, or modifying the exception's parameter.

Pattern MC-UP-CH: ***Update of Method Invocation in Catch Blocks*** This bug fix pattern consists in modifying the source code inside a catch body. This bug fix pattern hints that some bug fixes change the error handling code of *catch* blocks.

Pattern CONS-UP: ***Update of Super Constructor Invocation*** This bug fix pattern refers to the modification of *super* statement invocation, e.g., to change the parameter values. This

bug related to incorrect calls to *super* were so far not discussed. "Super" is, according to our teaching experience, a hard concept of object-oriented design.

Pattern IF-MC-ADD: ***Addition of Conditional Method Invocation*** This bug fix pattern adds an *if* whose block contains one method invocation. This change could correspond to the addition of a guarded invocation, typically done in a bug fix to add missing logic in a limit cases. In Pan et al.'s catalog, there is a pattern "Addition of Precondition Check", that only adds the guard around an existing block. In contrast, our pattern also specifies the addition of both the precondition and the code of the "if block". Consequently, both patterns are related, they share the same motivation, but they are conceptually disjoint.

Pattern IF-AS-ADD: ***Addition of Conditional Assignment*** This bug fix pattern represents the case of adding an *if* statement and an assignment inside its block. It corresponds to a modification of a variable value under a specific condition defined by the *if*.

3.4.5.3 Results

In this section, we present a formalization of bug fix patterns, using the formalization presented in section 3.4.2.2. We formalize: *a*) 18 bug fix patterns from Pan et al., belonging to the categories If, Loops, Try, Switch, Method Declaration and Assignment; *b*) 2 patterns proposed by Nath et al. [64]; *c*) 6 new ones presented in section 3.4.5.2.

Table 3.6 shows the result of the formalization of those bug fix patterns. The table groups the formalization according the source of the patterns i.e., Pan, Nath, and the new bug fix patterns presented in section 3.4.5.2. Column *Name* shows the bug fix pattern identifier. The remaining three columns correspond to the formalization itself: *L (Micro-Patterns)* the list of micro-patterns, *U (Undesired Micro-Patterns)* the list of undesired changes and *R (Relational Map* relations between micro-patterns.

The table presents bug fix patterns that are formalized by two or more sub-patterns. For example, pattern IF-APCJ (Addition of If PreCondition and Jump statement) is formalized by three sub-patterns. Each of these sub-patterns identifies pattern instances with a concrete jump statement. One corresponds to "break" jump statement, the other to "continue" jump statement and the last one to "return" statement.

The table also shows that the size of the micro-pattern list *L* varies between one and three. For those that *L* has two or three micro-patterns such as TY-ARTC, it exists a relation in *R* that defines a relation between micro-patterns (See Section 3.4.2.2.2).For those that *U* is not empty, a relation from *U* links a micro-pattern from *R* with another *U* (See Section 3.4.2.2.3).

In section 3.4.8 we discuss the limitations of our approach to formalize the remaining patterns from Pan et al. bug fix catalog.

3.4.5.4 Summary

In this section, we have shown that our approach is able to formalize 26 bug fix patterns. This answers our research question: our approach is flexible enough to formalize bug fix patterns from the literature and can also be used to specify new bug fix patterns.

```
   Input: L                                          ▷ List of micro-patterns
   Input: Changes                                    ▷ List of AST changes of a hunk
   Output: boolean value: true if all micro-patterns of the pattern are mapped to AST
           changes of the hunk, false otherwise
   Output: Mapping of Micro-Patterns and Changes from Changes
 1 begin
       /* Retrieves the first micro-pattern                                    */
 2     micro_pattern_i ← getMicropattern(L,0) ;
       /* Search AST changes of the hunk that matches with
          micro_pattern_i                                                      */
 3     initial_changes ← getFirstMatchingChanges(Changes, micro_pattern_i) ;
 4     if initial_changes is null then
 5         return false, ∅
 6     foreach change initial of the list initial_changes do
 7         change_i ← initial; M ← ∅ ;
 8         partialMapping ← true;
 9         cardinality_iter ← 0 ;
10         while partialMapping and micro_pattern_i is not null and change_i is not null do
               /* cardinality receives a natural number or * (wildcard)       */
11             cardinality ← cardinality(micro_pattern_i);
               /* Comparison of AST change and micro-pattern                   */
12             if match(micro_pattern_i, change_i) then
13                 saveMapping(M, micro_pattern_i, change_i) ;
14                 change_i ← getNextASTChange(Changes, change_i);
15                 cardinality_iter ←cardinality_iter +1 ;
16                 if cardinality != "*" and cardinality_iter == cardinality then
17                     micro_pattern_i ← getNextMicropattern(L, micro_pattern_i);
18                     cardinality_iter ← 0 ;
19             else
                   /* if current micro_pattern_i could be mapped, analyze
                      next micro-pattern                                       */
20                 if cardinality == "*" and isMapped(M, micro_pattern_i);
21                 then
22                     micro_pattern_i ← getNextMicropattern(L, micro_pattern_i);
23                     cardinality_iter ← 0 ;
24                 else
25                     partialMapping ← false;
           /* Return true if all analyzed changes are mapped and all
              micro-patterns from L are mapped to changes from
              Changes                                                          */
26         if partialMapping and allMapped(M, L, Changes) then
27             return true, M
28     return false, ∅
```

Figure 3.6: Algorithm to map micro patterns to AST changes

```
   Input: R                                          ▷ List of relations of a pattern
   Input: M                                  ▷ Mapping of Micro-Patterns and Changes
   Output: boolean value: true if the mapped AST changes respect the relations defined
           by the pattern, false otherwise
 1 begin
 2     foreach relation relation of the list R do
 3         micro_pattern_1 ← getFirstMicropattern(relation);
 4         micro_pattern_2 ← getSecondMicropattern(relation);
 5         changes_1 ← getMappedChanges(micro_pattern_1, M);
 6         changes_2 ← getMappedChanges(micro_pattern_2, M);
 7         foreach change change_1 of the list changes_1 do
 8             foreach change change_2 of the list changes_2 do
 9                 id_entity_1 ← getIdFromEntityType(relation.entity_1, change_1);
10                 id_entity_2 ← getIdFromEntityType(relation.entity_2, change_2);
                   /* operator receives values "==" or "!="                    */
11                 operator ← relation.comp;
                   /* Applies the comparison operator operator to
                      id_entity_1 and id_entity_2                              */
12                 comparison ← evaluate(id_entity_1, id_entity_2, operator) ;
                   /* If the relation is not valid, the phase returns
                      false                                                    */
13                 if comparison == false then return false;
       /* All relations were valid                                            */
14     return true;
   /* Return an entity identifier according to the field (et, pt
      and opt) that a relation links                                           */
15 Function(getIdFromEntityType(relation, change) : id)
16 begin
17     if relation.entity is a et field then
18         return change.id_et;
19     else
20         if relation.entity is a pt field then
21             return change.id_pt;
22         else
23             return change.id_opt;
```

Figure 3.7: Algorithm to verify the relation between AST changes

Name	L (Micro-Patterns)	U (Undesired Micro-Patterns)	R (Relational Map)	
New bug fix pattens				
DEC-RM	Delete of Variable declaration statement	mp1 = (Statement delete, Variable declaration, Method)		
THR-UP	Update of Throw statement	mp1 = (Statement update, Throw, Method)		
MC-UP-CH	Update of Method invocation in Catch clause	mp1 = (Statement update, Method invocation, Catch clause)		
CONS-UP	Update of Super constructor invocation	mp1 = (Statement update, Super constructor invocation, Method)		
IF-MC-ADD	Addition of precondition method invocation	mp1 = (Statement insert,If, Method) mp2 = (Statement insert, Method invocation, If)		mp2.pt == mp1.et
IF-AS-ADD	Addition of precondition assignment	mp1 = (Statement insert, If, Method) mp2 = (Statement insert, Assignment, If)		 mp2.pt == mp1.et
Pan et al. bug fix pattens				
IF-APC	Addition of Precondition Check	$mp1$ = (Statement Insert, If,*) $mp2$ = (Statement Parent Change,*, If)		mp2.pt == mp1.et
IF-APCJ	Add Precondition Check with Jump (3 subcases)	$mp1$ = (Statement Insert, If,*) $mp2$ = (Statement Insert, Break, If)		mp2.pt == mp1.et
		$mp1$ = (Statement Insert, If,*) $mp2$ = (Statement Insert, Continue, If)		
		$mp1$ = (Statement Insert, If,*) $mp2$ = (Statement Insert, Return, If)		
IF-RMV	Removal of an If Predicate	$mp1$ = (Statement Delete, If,*) $mp2$ = (Statement Parent Change,*, If)		mp2.pt == mp1.pt
IF-ABR	Addition of an Else Branch	$mp1$ = (Alternative Part Insert, Else Statement,*)	u_mp1 = (Statement Insert, If,*)	u_mp1.pt != mp1.pt
IF-RBR	Removal of an Else Branch	$mp1$ = (Alternative Part Delete, Else Statement,*) $mp2$ = (Statement Parent Change,*, Else Statement)	u_mp1 = (Statement Delete, If,*)	u_mp1.opt != mp1.et
IF-CC	Change of If Condition Expression	$mp1$ = (Condition Expression Change, If,*)		
LP-CC	Change of Loop Predicate(3 subcases)	$mp1$ = (Condition Expression Change, While,*)		
		$mp1$ = (Condition Expression Change, For,*)		
		$mp1$ = (Condition Expression Change, Do,*)		
SW-ARSB	Addition of Switch Branch	$mp1$ = (Statement Insert, Switch Case,*)	u_mp1 = (Statement Insert, Switch Statement,*)	u_mp1.et != mp1.pt

SW-ARSB	Removal of Switch Branch	$mp1$ = (Statement Delete, Switch Case,*)	u_mp1 = ((Statement Delete, Switch Statement,*)	u_mp1.et != mp1.pt
TY-ARTC	Addition of Try Statement	$mp1$ = (Statement Insert, Try,*) $mp2$ = (Statement Parent Change , *, Try) $mp3$ = (Statement Insert, Catch Clause,*)		$mp1.et == mp2.pt$ and $mp1.et == mp3.pt$
TY-ARTC	Removal of Try Statement	$mp1$ = (Statement Delete, Try,*) $mp2$ = (Statement Parent Change, *, Try) $mp3$ = (Statement Delete, Catch Clause,*)		$mp1.et == mp2.pt$ and $mp1.et == mp3.pt$
TY-ARCB	Addition of a Catch Block	$mp1$ = (Statement Insert, Catch Clause,*)	u_mp1 = (Statement Insert, Try,*)	u_mp1.et != mp1.pt
TY-ARCB	Removal of a Catch Block	$mp1$ = (Statement Delete, Catch Clause,*)	u_mp1 = (Statement Delete, Try,*)	u_mp1.et != mp1.pt
MD-CHG	Change of Method Declaration (7 subcases)	$mp1$ = (Parameter Insert, Single Variable Declaration,*)		
		$mp1$ = (Parameter Delete, Single Variable Declaration,*)		
		$mp1$ = (Parameter Type Change, Simple Type,*)		
		$mp1$ = (Parameter Type Change, Primitive Type,*)		
		$mp1$ = (Parameter Ordering Change, Single Variable Declaration,*)		
		$mp1$ = (Return Type Change, Simple Type,*)		
		$mp1$ = (Return Type Change, Primitive Type,*)		
MD-ADD	Addition of a Method Declaration	$mp1$ = (Additional Functionality, Method,*)		
MD-RMV	Removal of a Method Declaration	$mp1$ = (Removed Functionality, Method,*)		
CF-ADD	Addition of a Class Field	$mp1$ = (Additional Object State, Attribute,*)		
CF-RMV	Removal of a Class Field	$mp1$ = (Removed Object State, Attribute,*)		
Nath et al. bug fix pattens				
Nath-1	Scope changes (2 subcases)	$mp1$ = (Statement Parent Change, *,*,*)	u_mp1 = (Statement Delete, *,*) u_mp1 = (Statement Insert, *,*)	u_mp1.et != mp1.opt
		$mp1$ = (Statement Parent Change, *,*,*)		u_mp1.et != mp1.pt
Nath-3	Method return value changes	$mp1$ = (Statement Update, Return,*)		

Table 3.6: Formalization of bug fix patterns.

3.4.6 Evaluating the Accuracy of AST-based Pattern Instance Identifier

In this section, we present an experiment to measure the accuracy of our bug fix pattern instance identifier presented in section 3.4.4 . This evaluation is built on the following research question: *Is our AST-based pattern instance identifier more accurate than the state-of-the-art pattern identifier presented by Pan et al. [9]?*

The experiment consists of a manual inspection and validation of bug fix pattern instances identified in commits of an open-source project. Given a bug fix commit and an instance of pattern P_i identified by an identifier, the instance is considered valid if the manual inspection validates that the change indeed corresponds to pattern P_i. Otherwise, the instance is considered invalid.

3.4.6.1 Evaluated Pattern Instance Identifiers

3.4.6.1.1 Baseline Classifier The baseline tool we selected is called SEP[16]. SEP is a token-based classifier used to identify bug fix instances from revisions of Java files (a revision is a pair of file, say Foo.java version 1.1 and Foo.java version 1.2). According to the code symbols, this tool was used to gather the results presented in Pan et al.'s study [9].

3.4.6.1.2 AST-based Classifier We develop a tool that implements the AST classifier presented in Section 3.4.4. The tool is implemented in Java and uses ChangeDistiller [22] to obtain AST-level differences between consecutive revisions of a file. We use a publicly available implementation of ChangeDistiller[17].

We limit both tools to identify instances of 18 bug fix patterns from Pan et al. bug fix catalog. These patterns are those we are able to represent using our pattern formalization presented in section 3.4.5.3.

3.4.6.2 Analyzed Data

We randomly selected a sample of 86 revisions (pairs of Java files) from the CVS history of the Lucene open-source project (from 09/2001 to 02/2006). The sampling strategy is that those revisions contain a small number of source code changes, less than 5 AST changes (this excludes formatting and documentation changes). Lucene is one of the six open-source software applications used in Pan et al.'s work. The dataset is available on `https://sites.google.com/site/matiassebastianmartinez/journal.zip`.

3.4.6.3 Experimental Results

Table 3.7 shows the result of the manual inspection for pattern instances from Lucene's revisions identified by our AST-based approach and SEP tool. For each algorithm, the table shows the number of valid pattern instances, i.e. the true positives (column "Valid") and the number of invalid instances, i.e., the false positive instances (column "Not Valid"). Moreover, it shows a number of missing instances (false negatives) i.e. valid instances that an approach could identify but the other could not (column "Missing"). This number is not an

[16] http://gforge.soe.ucsc.edu/gf/project/sep/scmsvn/
[17] http://www.ifi.uzh.ch/seal/research/tools/changeDistiller.html

	Valid	Not Valid	Missing
Pan et al's Token-based Approach	62	74	27
Our AST-based Approach	78	0	11

Table 3.7: The Results of the Manual Inspection of Bug Fix Pattern Instances. The row "Token-Based" corresponds to the instances identified by the token-based classifier. The row "AST-Based" corresponds to the instances identified by the AST classifier.

absolute number of false negatives, it is only relative with respect to the approach. In the remaining of this section we study the accuracy of both approaches.

3.4.6.3.1 Accuracy Definition We define the accuracy of a bug fix pattern identifier as follows:

$$accuracy = \frac{\text{number of bug fix instance correctly identified}}{\text{total number of instance identified} + \text{missing pattern instance}}$$

For instance, a pattern identifier that identifies 5 instances, all correctly, but misses 2 instances, has an accuracy of $5/(5+2) = 0.71$. Another example is an identifier that correctly identifies 4 instances, incorrectly 1 and misses 2. Its accuracy is $4/(4+1+2) = 0.57$. According to the accuracy values, the first identifier is *more accurate* than the second one.

3.4.6.3.2 Accuracy of Token-based Identification The token-based identifier finds 136 instances of bug fix patterns. Table 3.7 shows that our manual inspection found 62 valid pattern instances (true positives), 74 invalid (false negatives) and 27 missing instances. The accuracy of the token-based instance identifier is $62/(62+74+27) = 0.38$.

Let us analyze some cases where the identifier finds invalid instances. For instance, the token-based identifier identifies from revision 1.4 of class "FilteredQuery", an invalid instance of pattern "Change of Method Declaration" (MD-CHG) and another invalid instance of pattern "Addition of precondition with jump" (IF-APCJ). The actual bug fix pattern in this commit is "Addition of Method Declaration" (MD-ADD). The first invalid instance is due to a wrong mapping between the lines of the revision. The added method is "mapped" to an existing method (with different signature), resulting in the change being interpreted as an update of the method declaration. For the second false positive, the invalid pattern instance is identified inside the code of the added method.

We also found false positive instances caused by formatting changes between consecutive revisions. For example, the revision 1.3 of Lucene's class GermanStemmer applies formatting changes in the source code and among the many modified lines, one local variable is initialized. The token-based identifier incorrectly identifies from this pair 21 instances of 9 different bug fix patterns. The formatting changes produce a complex mapping between the revision and its predecessor in many hunks. Consequently, the code inside these formatting hunks matching with a bug fix pattern definition is incorrectly identified as an instance.

3.4.6.3.3 Accuracy of AST-based Identification The AST-based identifier found 78 bug fix pattern instances. These instances were present in 53 different revisions. Moreover, the

identifier could not identify 11 instances (missing). The accuracy of this identifier is $78/(78+11) = 0.88$.

Our manual inspection found that 100% (78/78) of the pattern instances were valid (the data is available). This implies all bug fix instances were *true positives*. These identified instances correspond to 9 different bug fix patterns. However, the algorithm also missed some instances, i.e., suffers from false negatives, which are discussed in Section 3.4.6.3.4.

3.4.6.3.4 False Negatives A false negative (or missing) instances is a valid bug fix pattern instance which is not identified by a pattern instance identifier. To detect those instances from the analyzed data, we cross the results obtained from both AST and token-based approaches. A missing instance of a pattern instance identifier A is not identified by A but is identified correctly by the other approach.

Table 3.7 presents the classification result. The token-based approach had 27 missing bug fix instances while the AST-based one had 11 false negatives.

Let us now analyze the false negatives of our approach. For 7 of 11 missing instances, the cause is due to the tree differencing tool (ChangeDistiller) we use to compute the differences between two consecutive revisions. ChangeDistiller does not compute changes inside anonymous and inner classes and there were 7 pattern instances in such classes in our data. For example, our algorithm does not identify an instance of pattern "Removal of if predicate (IF-RMV)" in revision 1.21 of class IndexSearcher, the instance is in the inner class HitCollector. Another case is that our approach does not see changes in the specification of thrown exceptions (keyword "throws" in Java), which are instances of pattern "Change of method declaration (MD-CHG)". For example, revision 1.4 of class TestTermVectorsWriter modifies the signature of the method by adding a clause "throws IOException". Our tree differencing algorithm does not consider those changes and this limitation impacts the accuracy of this particular bug fix pattern.

3.4.6.4 Conclusion

The manual analysis done in the presented experiment allows us to respond to our research question: our AST-based identifier is more accurate than the token-based used by Pan et al. in their experiments.

The results of our experiment are summarized in Table 3.7. It shows that our AST-based identifier is able to identify: more valid bug fix instances (more true positives); less invalid instances (less false positives); less number of missing instances (less false negatives). Consequently, we can say that it is more accurate (0.88 vs. 0.38) than the token-based approach.

3.4.7 Learning the Abundance of Bug Fix Patterns

In this section, we use the bug fix pattern instance identifier presented in Section 3.4.4 to measure the abundance of bug fix patterns. The abundance allows us to measure the importance of bug fix patterns. Then, one can define a probabilistic repair model formed of bug fix patterns and their frequencies.

This kind of probabilistic repair model could be used by bug fix pattern-based repair approaches such as PAR [5]. Let us explain how using PAR approach as example. PAR is a repair approach guided by evolutionary computation. To create candidate fixes, it instantiates 10 bug fix templates, derived from bug fix patterns. PAR navigates the search space in a

uniform random way, that means, it takes randomly one bug fix template to be applied in a buggy location. The pattern abundance could be used in an extension of the strategy to navigate the search space. Instead of a random strategy, the extension could start navigating the space from the most abundant bug fix templates (i.e., the most frequent kind of fixes applied by developers) to the less abundant. This strategy could help to find a fix faster, avoiding applying infrequent changes in bug fixing.

	#Commits	#Revisions	#Java Revisions
All	24,042	173,012	110,151
BFP	6,233	33,365	23,597

Table 3.8: Versioning data used in our experiment. Since we focus on bug fix patterns, we analyze the 23,597 Java revisions whose commit message contains "bug", "fix" or "patch".

3.4.7.1 Dataset

We have searched for instances of the 18 patterns mentioned in 3.4.5 in the history of six Java open source projects: ArgoUML, Lucene, MegaMek, Scarab, jEdit and Columba. In Table 3.8 we present the total number of commits (versioning transactions) and revisions (file pairs) present in the history of these projects. In the rest of this section, we analyze the 23,597 Java revisions whose commit message contains "bug", "fix" or "patch", in a case insensitive manner (row "BFP" in Table 3.8).

3.4.7.2 Empirical Results

Table 3.9 shows that our approach based on AST analysis scales to the 23,597 Java revisions from the history of 6 open source projects. This table enables us to identify the importance of each bug fix pattern. For instance, adding new methods (MD-ADD) and changing a condition expression (IF-CC) are the most frequent patterns while adding a try statement (TY-ARTC) is a low frequency action for fixing bugs. Overall, the distribution of the pattern instances is skewed, and it shows that some of Pan's patterns are really rare in practice. Interestingly, we have also computed the results on all revisions – with no filter on the commit message – and the distribution of patterns is rather similar. It seems that the bug-fix-patch heuristic does not yield a significantly different set of commits.

3.4.7.3 Summary

In this subsection, we presented the abundance of 18 bug fix patterns from the analysis of 6 open-source projects. We found that the most frequent changes to fix bugs are changes in *if condition* statements. Knowing this distribution is important in some contexts. For instance, from the viewpoint of automated software repair approaches: their fix generation algorithms can concentrate on likely bug fix patterns first in order to maximize the probability of success.

Pattern name	Abs
Change of If Condition Expression-IF-CC	4,444
Addition of a Method Declaration-MD-ADD	4,443
Addition of a Class Field-CF-ADD	2,427
Addition of an Else Branch-IF-ABR	2,053
Change of Method Declaration-MD-CHG	1,940
Removal of a Method Declaration-MD-RMV	1,762
Removal of a Class Field-CF-RMV	983
Addition of Precond. Check with Jump-IF-APCJ	667
Addition of a Catch Block-TY-ARCB	497
Addition of Precondition Check-IF-APC	431
Addition of Switch Branch-SW-ARSB	348
Removal of a Catch Block-TY-ARCB	343
Removal of an If Predicate-IF-RMV	283
Change of Loop Predicate-LP-CC	233
Removal of an Else Branch-IF-RBR	190
Removal of Switch Branch-SW-ARSB	146
Removal of Try Statement-TY-ARTC	26
Addition of Try Statement-TY-ARTC	18
Total	21,234

Table 3.9: Context-independent Bug Fix Patterns: Absolute Number of Bug Fix Pattern Instances in 23,597 Java Revisions.

3.4.8 Discussion

3.4.8.1 Threats to Validity

Our results are completely computational and a severe bug in our implementation may invalidate our findings. During our experiments, we studied in details dozens of bug fix pattern instances (the actual code, the fix and the commit message) found by the tool and they were meaningful.

Another threat is the criterion to manually classify a bug fix pattern instance as valid or not. It could vary depending on who inspects it (an expert, a developer, a novice, etc.).

3.4.8.2 Limitations

In this section we sum up the limitations of our bug fix pattern formalization approach.

3.4.8.2.1 Context Dependence We notice that some patterns only describe the nature of change itself, while others describe the change in a given context. By nature of change, we mean only the added and removed content; by context, we mean the code around the added and removed content. For instance, Pan et al. define a pattern representing the removal of a method call in a sequence of method calls. To us, this pattern is *context-dependent*. To observe an instance of removal of a method call in a sequence of method calls: 1) the change itself has to be a removal of a method call 2) the context of the removal has to be a sequence of method calls on the same object. In total, there is a minority of 8/31 bug fix patterns of the refined catalog presented in section 3.4.5.1 that are context-dependent.

We do not consider those context-dependent bug fix patterns. This limitation could probably be overcome with a way to specify the "context" (the code surrounding the diff) at the AST level.

3.4.8.2.2 Limitations inherited from ChangeDistiller Another limitation of our pattern formalization approach is due to the change taxonomy used by tree differencing algorithm. ChangeDistiller misses some kinds of source code changes. For instance, an update operation in a class field declaration is not detected. This limitation prevents us to represent pattern "Change of Class Field Declaration" (CF-CHG) using AST changes as well as 4 other patterns. Those 5 patterns contain at least one change that is not covered by the change taxonomy of ChangeDistiller.

Another limitation is the granularity of the tree differencing algorithm. ChangeDistiller works at the statement level. This prevents us to study certain fine-grain patterns. For example, the addition of a new parameter or the change of an expression passed as parameter of a method call cannot be detected. Also, as we discussed in Section 3.4.6.3.4, the tree differencing algorithm does not detect changes inside anonymous classes. Improvement or replacement of the tree differencing algorithm could potentially decrease the number of false negatives.

3.5 Recapitulation

In this chapter, our challenge was to learn how developers fixed bugs. There is information from them that could be useful in automatic software repair domain. We first studied how

software evolves. In particular, we studied all changes that developers do in the history on a set of open-source software. Furthermore, we studied the importance of source code changes. We learned that there are changes that appear more frequent than others in the software evolution.

Then, we focused in how software is repaired. We first presented a technique to recognize repairs from the history of a program. Then, we analyzed the composition of these repairs in two levels of granularity: at the AST level and a change pattern level. We proposed mechanism to measure the importance of those levels.

The contributions of this chapter are:

1. One technique to filter repairs from version control system.
2. Two models that describe the kind of source code changes done by developers to fix bugs, and include a measure of the importance of those changes.
3. One mechanism to formalize change patterns.
4. One mechanism to identify pattern instances using the formalization.

The work presented in this section has direct applications. For instance, one is for the repair approaches evaluation. It allows to define a fair and unbiased defect dataset for approaches evaluation (Chapter 5). An evaluation dataset that targets a defect class (defined further) can be composed of instances of bug fix patterns identified by our mechanism. Moreover, this work can be used for the improvement of software repair approaches (Chapter 4).

Chapter 4

Two Strategies to Optimize Search-based Software Repair

In Chapter 3 we presented models that characterize the behavior of developers fixing bugs. In this Chapter, we aim at studying repair search spaces built from those repairs models.

The search space of automated program repair consists of all explorable bug fixes for a given program and a given bug. A naive search space is huge, because even in a bounded size scenario, there are a myriad of elements to be added, removed or modified: statements, variables, operators, literals. The search of the solution, i.e., a bug fix, is guided by a *search strategy*. For instance, GenProg [12] uses a fix search guided by genetic programming, while Qi et al.[76] use random search. The time those strategies spend to navigate the search space could be infinite, especially when the size of the search space is huge. As consequence, the search of the fix is delimited by some criteria such as the time or number of candidate repairs validated. For example, the search in GenProg is delimited by the number of program variants (each with one candidate fix) to validate. This number depends on two variables of the genetic programming search: size of the initial population and number of generations to evolve each member of the population. For instance, in one of its evaluations [99], GenProg executes 400 variants (from an initial population of 40 and 10 generations).

Unfortunately, during repair approach evaluations, there are bugs that remain unrepairable. For example, GenProg [8] is able to repair 55 out of 105 bugs. We have two hypotheses about the reason a solution is not found. The first one states the repair approach is not able to fix the bug. This means, its repair search space does not contain the fix. The second one states that at least one fix exists in the solution space but it was not discovered by the search strategy.

In this chapter we focus on the latter hypothesis. We aim at defining strategies to find those undiscovered solutions in the repair search space. These strategies focus on the way a search space is navigated.

We present two navigation strategies. The first strategy, presented in Section 4.1, takes as input human bug fixes. The strategy aims at navigating the search space in the following manner: the probability to select a fix from the space depends on the frequency this kind of fixes is used by developers to fix. By focusing on frequent repairs, the strategy aims at reducing the time to fix a bug and, by consequence, to increase the probability of finding a repair.

The second strategy, presented in Section 4.2 aims at improving the search space navigation of redundancy-based repair approaches such as GenProg. Redundancy-based repair approaches synthesize fixes reusing already written source code. Our strategy aims at reducing the locations where the reusable code is picked to synthesize a fix. As consequence, the strategy produces a smaller search space without losing repair strength. That means, the reduced space contains as much solutions as the original (not reduced) space.

This chapter contains material published in the proceedings of ESEM'13 [11], ICSE'14 [100] and unpublished material.

4.1 Adding Probabilities to the Search Space to Optimize the Space Navigation

This section discusses the nature of the search space size of automated program repair. In Section 3.1.2 we defined two change models, CT and CTET, and we showed in Section 3.3 that both models can be extended adding probabilistic distribution over their repair actions.

In this Section we present a strategy to optimize the navigation of the search space. Our challenge is to know whether exists a repair model that allows repair approaches to navigate the shaping phase faster than others, in other words, in a more efficient way. This allows repair approach to increase the probability to find a fix.

The Section remains as follows. In Section 4.1.1 we present a typical composition of repair search spaces. In Section 4.1.2 we present a strategy to reduce the time for searching elements in the search space. In Section 4.1.3 we present the evaluation of the strategy. Section 4.1.3.3 presents a theoretical study case that includes probabilistic search space over a repair approach from the literature. Finally, Section 4.1.4 concludes the section.

4.1.1 Decomposing Repair Search Space

We consider that the repair search space can be viewed as the combination of three spaces: the fault localization space, the shaping space, and the synthesis space.

The search space can then be loosely defined as the Cartesian product of those spaces and its size then reads:

$$|\text{FAULT LOCALIZATION}| \times |\text{SHAPE}| \times |\text{SYNTHESIS}|$$

The fault localization space contains the location where a repair is likely to be successful. The shaping space contains the kind of repair that can be applied. Informally, the shape of a bug fix is a kind of patch. For instance, the repair shape of adding an "if" throwing an exception for signaling an incorrect input consists of inserting an if and inserting a throw. The concept of "repair shape" is equivalent to what Wei et al. [74] call a "fix schema", and Weimer et al. [12] a "mutation operator". We define a "repair shape" as an unordered tuple of repair actions (from a set of repair actions called $\mathcal{R}$, see Section 3.3)[18]. For the if/throw example aforementioned, in repair space CTET, the repair shape of this bug fix consists of two repair actions: 1) statement insertion of "if" and 2) statement insertion of "throw". The shaping space consists of all possible combinations of repair actions.

[18]Since a bug fix may contain several instances of the same repair actions (e.g. several statement insertions), the repair shape may contain several times the same repair action.

Finally, the synthesis space contains the initializations of the repair shapes. The complexity of the synthesis depends on the repair actions of the shaping space. For instance, the repair actions of Weimer et al. [12] (insertion, deletion, replace) have an "easy" and bounded synthesis space. The approach does not synthesize new code. Instead, it instantiates insertion and replace repair actions with code that already exists in the program.

In this section we present a strategy to decrease the navigation time of a shaping search space. If one can find efficient strategies to navigate through this shaping space, this would contribute to efficiently navigating through the repair search space as a whole, thanks to the combination.

4.1.2 A Strategy to Optimize Shaping Space Navigation

We present a strategy to optimize the navigation of the shaping space. The strategy aims at finding faster a fix from the search space. For that, it aims at exploring the parts of the space that are likely to be more fruitful. The strategy relies on probability repair models presented in Section 3.3: a repair action is selected from the space according to its probability distribution. That means that the probabilistic distributions $\mathcal{P}$ over the repair actions *guide* the navigation of the repair space. As a consequence, the probability distribution is crucial for minimizing the search space traversal: a good distribution $\mathcal{P}$ results in *concentrating on likely repairs first*. We aim at providing the search strategy a probability distribution $\mathcal{P}_{min}$ allows minimizing the time of navigating the search space. The challenge of this section is to find a way to compare probabilistic distributions for finding $\mathcal{P}_{min}$. For that, we need to: *a*) estimate the navigation time of the shaping space for a given probability distribution over a repair model; *b*) set up a probability distribution over repair actions; *c*) compare the efficiency of different probability distributions to find good repair shapes. In the remaining of this section we target these points.

4.1.2.1 Mathematical Analysis Over Repair Models

To analyze the shaping space, we now present a mathematical analysis of our probabilistic repair models. So far, we have two repair models CT and CTET (see 3.3) and different ways to parametrize them.

According to our probabilistic repair model, a good navigation strategy consists on concentrating on likely repairs first: the repair shape is more likely to be composed of frequent repair actions. That is a repair shape of size n is predicted by drawing n repair actions according to the probability distribution over the repair model. Under the pessimistic assumption that repair actions are independent[19], our repair model makes it possible to know the exact median number of attempts N that is needed to find a given repair shape R (demonstration given in A):

$$N = k \text{ such that } \sum_{i=1}^{k} p(1-p)^{i-1} \geq 0.5 \tag{4.1}$$

[19]Equation (1) holds if and only if we consider them as independent. If they are not, it means that we underestimate the deep structure of the repair space, hence we over-approximate the time to navigate in the space to find the correct shape. In other words, even if the repair actions are not independent (which is likely for some of them) our conclusions are sound.

$$\text{with } p = \frac{n!}{\Pi_j(e_j!)} \times \Pi_{r \in R} P_{\mathcal{P}}(r)$$

where e_j is the number of occurrences of r_j inside R

For instance, the repair of revision 1.2 of Eclipse's CheckedTreeSelectionDialog[20] consists of two inserted statements. Equation 4.1 tells us that in repair model CT, we would need in average 12 attempts to find the correct repair shape for this real bug.

Having only a repair shape is far from having a real fix. However, the concept of repair shape associated with the mathematical formula analyzing the time to navigate the repair space is key to compare ways to build a probability distribution over repair models.

Input: C ▷ A bag of transactions
Output: The median number of attempts to find good repair shapes
1 **begin**
2 $\Omega \leftarrow \{\}$ ▷ Result set
3 $T, E \leftarrow split(C)$ ▷ Cross-validation: split C into *T*raining and *E*valuation data
4 $M \leftarrow train\_model(T)$ ▷ Train a repair model (e.g. compute a probability distribution over repair actions)
5 **for** $s \in E$ ▷ For all repairs observed in the repository
6 **do**
7 $n \leftarrow compute\_repairability(s, M)$ ▷ How long to find this repair according to the repair model
8 $\Omega \leftarrow \Omega \cup n$ ▷ Store the "repairability" value of s
9 **return** $median(\Omega)$ ▷ Returning the median number of attempts to find the repair shapes

Figure 4.1: An Algorithm to Compare Fix Shaping Strategies. There may be different flavors of functions $split$, f and $computeRepairability$.

4.1.2.2 Defining Probabilistic Repair Models

To compute a probability distribution over repair actions, we propose to learn them from software repositories. For instance, if many bug fixes are made of inserted method calls, the probability of applying such a repair action should be high. Despite our single method (learning the probability distributions from software repositories), we have shown in 3.3 that there is no single way to compute them, they depend on different heuristics. In the evaluation (Section 4.1.3), we use those heuristics BFP and N-SC to define shaping spaces.

4.1.2.3 Comparing Different Distributions

To compare different distributions against each other, we set up the following process. One first selects bug repair transactions in the versioning history. Then, for each bug repair transaction, one extracts its repair shape (as a set of repair actions of a repair model). Then one

[20] "Fix for 19346 integrating changes from Sebastian Davids" `http://goo.gl/d4OSi`

computes the average time that a maximum likelihood approach would need to find this repair shape using equation 4.1.

Let us assume two probability distributions $\mathcal{P}_1$ and $\mathcal{P}_2$ over a repair model and four fixes ($F_1 \dots F_4$) consisting of two repair actions and observed in a repository. Let us assume that the time (in number of attempts) to find the exact shape of $F_1 \dots F_4$ according to $\mathcal{P}_1$ is $(5, 26, 9, 12)$ and according to $\mathcal{P}_2$ $(25, 137, 31, 45)$. In this case, it is clear that the probability distribution $\mathcal{P}_1$ enables us to find the correct repair shapes faster (the shaping time for $\mathcal{P}_1$ is lower). Beyond this example, by applying the same process over real bug repairs found in a software repository, our process enables us to select the best probability distributions for a given repair model.

Since equation 4.1 is parametrized by a number of repair actions, we instantiate this process for all bug repair transactions of a certain size (in terms of AST changes). This means that our process determines the best probability distribution for a given bug fix shape size.

4.1.2.3.1 Comparison Algorithm Figure 4.1 sums up this algorithm to compare fix shaping strategies. The algorithm uses cross-validation to avoid bias in the result. This bias can emerge due we use the same data, i.e., transactions found in repositories, to: compute different probability distributions $\mathcal{P}_x$, and to evaluate the time to find the shape of real fixes (bug fix transactions). To overcome this problem, we always use different sets of transactions to estimate $\mathcal{P}$ and to calculate the average number of attempts required to find a correct repair shape. Using cross-validation reduces the risk of overfitting. Let us analyze the algorithm. From a bag of transactions C, function $split$ (line 3) creates a set of testing transactions and a set of evaluation transactions. Then, one trains a repair model, with function $trainModel$ (line 4), for repair models CT and CTET it means computing a probability distribution on a specific bag of transactions. Finally, for each repair of the testing data, one computes its "repairability" according to the repair model, with Equation 4.1 (line 7). The algorithm returns the median repairability, i.e., the median number of attempts required to repair the test data (line 9).

4.1.3 Evaluation

4.1.3.1 Evaluation set up

We run our fix shaping process on our dataset of 14 repositories of Java software considering two repair models: CT and CTET (see Section 3.1.2). We remind that CT consists of 41 repair actions and CTET of 173 repair actions. For both repair models, we have tested the different heuristics of 3.3.1 to compute the median repair time: all transactions (ALL); one AST change (1-SC); 5 AST changes (5-SC); 10 AST changes (10-SC); 20 AST changes (20-SC); transactions with commit text containing "bug", "fix", "patch" (BFP); a baseline of a uniform distribution over the repair model (EQP for equally-distributed probability).

Since we have a dataset of 14 independent software repositories, we use this dataset structure for cross-validation. We take one repository for extracting repair shapes and the remaining 13 projects to calibrate the repair model (i.e. to compute the probability distributions). We repeat the process 14 times, by testing each of the 14 projects separately. In other words, we try to predict real repair shapes found in one repository from data learned on other software projects.

We extracted all bug fix transactions with less than 8 AST changes from our dataset. For instance, the versioning repository of DNSJava contains 165 transactions of 1 repair action, 139 transactions of size 2, 71 transactions of size 3, etc. The biggest number of available repair tests are in jdt.core (1,605 fixes consist of one AST change), while Jhotdraw has only 2 transactions of 8 AST changes. We then computed the median number of attempts to find the correct shape of those 23,048 fix transactions. Since this number highly depends on the probability distributions $\mathcal{P}_x$, we computed the median repair time for all combinations of fix size transactions, projects, and heuristics discussed above $(8 \times 14 \times 6)$.

Repair /Repair Size	1	2	3	4	5	6	7	8
ArgoUML	**6** (996)	**13** (638)	**86** (386)	**267** (362)	**1394** (254)	**5977** (234)	**16748** (197)	**73430** (166)
Carol	**7** (30)	**13** (15)	**121** (10)	**466** (10)	**494** (7)	**24117** (13)	**14019** (6)	**30631** (9)
Columba	**3** (382)	**13** (255)	**68** (144)	**552** (146)	**940** (113)	**2111** (108)	**10908** (73)	**64606** (94)
Dnsjava	**6** (165)	**13** (139)	**101** (71)	**218** (82)	**1553** (54)	**5063** (50)	**16363** (33)	∞(44)
jEdit	**3** (115)	**13** (84)	**58** (53)	**251** (48)	**2906** (32)	**3189** (30)	**5648** (29)	**23395** (32)
jBoss	**6** (514)	**15** (353)	**88** (208)	**272** (189)	**1057** (147)	**6034** (150)	**13148** (86)	**38485** (113)
jHotdraw6	**7** (21)	**13** (21)	**159** (9)	**187** (10)	**1779** (10)	**611** (3)	∞(5)	**56391** (2)
jUnit	**3** (40)	**42** (39)	**596** (18)	∞(11)	**49345** (7)	∞(11)	**31634** (9)	∞(6)
Log4j	**6** (223)	**15** (134)	**146** (68)	**665** (70)	**6459** (64)	**16879** (42)	**55582** (41)	∞(48)
org.eclipse.jdt.core	**6** (1606)	**26** (1025)	**93** (657)	**291** (631)	**1704** (392)	**4639** (416)	**18344** (314)	**74863** (309)
org.eclipse.ui.workbench	**3** (1184)	**13** (783)	**74** (414)	**311** (464)	**1023** (326)	**6035** (305)	**22864** (215)	**77532** (192)
Scarab	**6** (653)	**16** (346)	**113** (202)	**420** (159)	**764** (113)	**3914** (137)	**13104** (89)	**59232** (77)
Struts	**3** (221)	**17** (133)	**100** (86)	**222** (103)	**675** (61)	**4785** (77)	**16796** (39)	**95588** (34)
Tomcat	**3** (281)	**13** (167)	**135** (111)	**431** (120)	**1068** (84)	**3497** (87)	**7407** (61)	**34240** (51)

Table 4.1: The median number of attempts (in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CT. The repair model CT is made from the distribution probability of changes included in 5-SC transaction bags. For small transactions, finding the correct repair shape in the search space is done in less than 100 attempts.

4.1.3.2 Empirical Results

Table A.17 presents the results of this evaluation for repair space CT and transaction bag 5-SC. For each project, the bold values give the median repairability in terms of number of attempts required to find the correct repair shape with a maximum likelihood approach. Then, the bracketed values give the number of transactions per transaction size (size in number of AST changes) and per project. For instance, over 996 fix transactions of size 1 in the ArgoUML repository, it takes an average of 6 attempts to find the correct repair shape. On the contrary, for the 51 transactions of size 8 in the Tomcat repository, it takes an average of 34,240 attempts to find the correct repair shape. Those results are encouraging: for small transactions, it takes a handful of attempts to find the correct repair shape. The probability distribution over the repair model seems to drive the search efficiently. The other heuristics yield similar results – the complete results (6 tables – one per heuristic) are given in appendix A.

About cross-validation, one can see that the performance over the 14 runs (one per project) is similar (all columns of Table A.17 contain numbers that are of similar order of magnitude). Given our cross-validation procedure, this means that for all projects, we are able to predict the correct shapes using only knowledge mined in the other projects. This gives us confidence that one could apply our approach to any new project using the probability distributions mined in our dataset.

Furthermore, finding the correct repair shapes of larger transactions (up to 8 AST changes) has an order of magnitude of 10^4 and not more. Theoretically, for a given fix shape of n AST changes, the size of the repair model is the number of repair actions of the model at the power of n (e.g. $|CT|^n$). For CT and $n = 4$, this results in a space of $41^4 = 2{,}825{,}761$ possible shapes (approx 10^6). In practice, overall all projects, for small shapes (i.e. less or equal than 3 changes), a well-defined probability distribution can guide to the correct shape in a median time lower than 200 attempts. This again shows that the probability distribution over the repair model is so unbalanced that the likelihood of possible shapes is concentrated on less than 10^4 shapes (i.e. that the probability density over $|CT|^n$ is really sparse).

Now, what is the best heuristic, with respect to shaping, to train our probabilistic repair models?

For each repair shape size of Table A.17 and heuristic, we computed the median repairability over all projects of the dataset (a median of median number of attempts). We also compute the median repairability for a baseline of a uniform distribution (EQP) over the repair model (i.e. $\forall i, P(r_i) = 1/|CT|$)). Figure 4.2 presents this data for repair model CT. It shows the median number of attempts required to identify correct repair shapes as Y-axis. The X-axis is the number of repair actions in the repair test (the size). Each line represents probability estimation heuristics.

Figure 4.2 gives us important pieces of information. First, the heuristics yield different repair time. For instance, the repair time for heuristic 1-SC is generally higher than for 20-SC. Overall, there is a clear order between the repairability time: for transactions with less than 5 repair actions heuristic 5-SC gives the best results, while for bigger transactions 20-SC is the best. Interestingly, certain heuristics are inappropriate for maximum-likelihood shaping of real bug fixes: the resulting distributions of probability results in a repair time that explodes even for small shape (this is the case for a uniform distribution EQP even for shape of size 3). Also, all median repair times tend toward infinity for shape of size larger than 9. Finally, although 1-SC is not good over many shape size, we note that for small shape of size 1 is

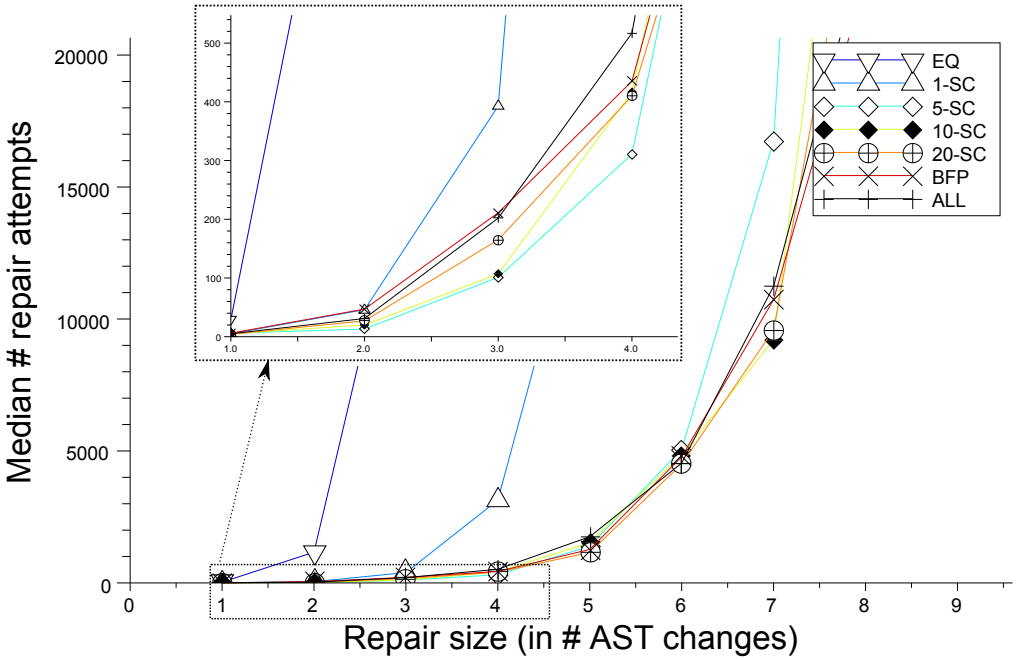

Figure 4.2: The repairability of small transactions in repair model CT. Certain probability distributions yield a median repair time that is much lower than others.

better. This is explained by the empirical setup (where we also decompose transactions by shape size).

4.1.3.2.1 On The Best Heuristics for Computing Probability Distributions over Repair Actions To sum up, for small repair shapes heuristic 1-SC is the best with respect to probabilistic repair shaping, but it is not efficient for shapes of size greater than two AST-level changes. Heuristics 5-SC and 20-SC are the best for changes of size greater than 2. An important point is that *some probability distributions (in particular built from heuristics EQP and 1-SC) are really suboptimal for quickly navigating into the search space*.

Do those findings hold for repair model CTET, which has a finer granularity?

4.1.3.2.2 On The Difference between Repair Models CT and CTET We have also run the whole evaluation with the repair model CTET (see 3.1.2). The empirical results are given in appendix A (in the same form as Table A.17).

Figure 4.3 is the sibling of figure 4.2 for repair model CTET. They look rather different. The main striking point is that with repair model CTET, we are able to find the correct repair shape for fixes that are no larger than 4 AST changes. After that, the arithmetic of very low probability results in virtually infinite time to find the correct repair shape. On the contrary, in the repair model CT, even for fixes of 7 changes, one could find the correct shape in a finite number of attempts. Finally, in this repair model the average time to find a correct repair shape is several times larger than in CT (in CT, the shape of fixes of size 3 can be found in approx. 200 attempts, in CTET, it's more around 6,000).

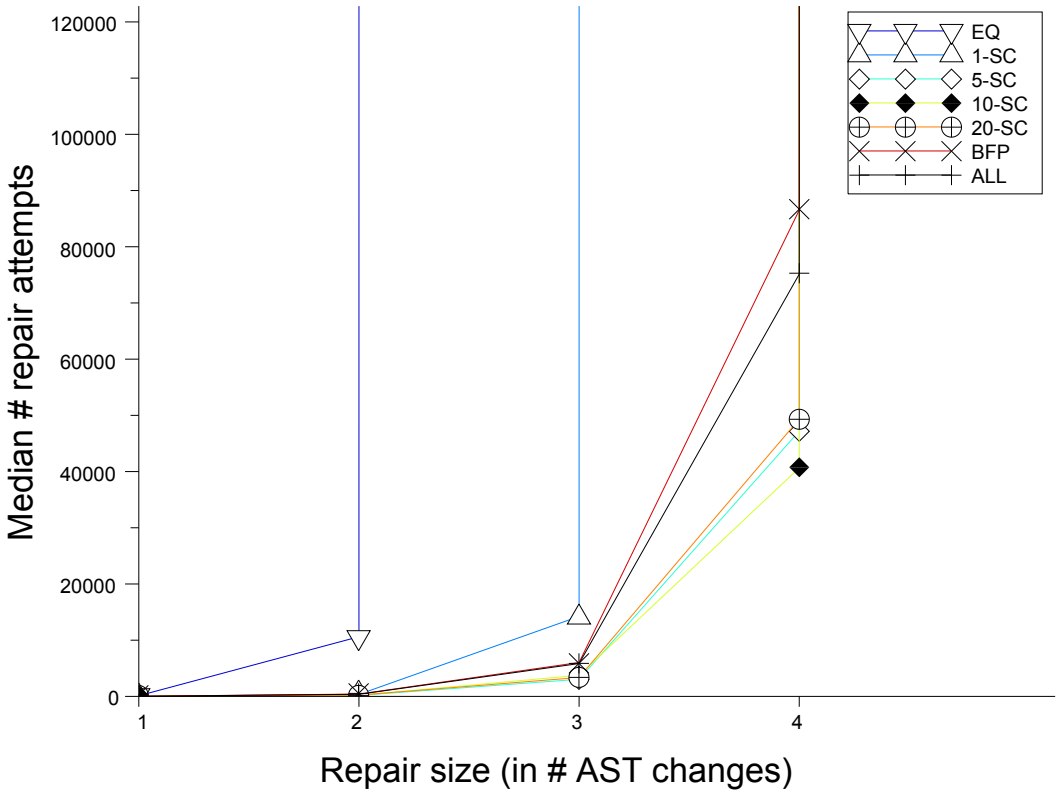

Figure 4.3: The repairability of small transactions in repair space CTET. There is no way to find the repair shapes of transactions larger than 4 AST code changes.

For a given repair shape, the synthesis consists of finding concrete instances of repair actions. For instance, if the predicted repair action in CTET consists of inserting a method call, it remains to predict the target object, the method and its parameters. We can assume that the more precise the repair action, the smaller the "synthesis space". For instance, in CTET, the synthesis space is smaller compared to CT, because it only composed of enriched versions of basic repair actions of repair model CT (for instance inserting an "if" instead of inserting a statement).

Our results illustrate the tension between the richness of the repair model and the ease of fixing bugs automatically. When we consider CT, we find likely repair shapes quickly (less than 5,000 attempts), even for large repair, but to the price of a larger synthesis space. In other words, there is a balance between finding correct repair actions and finding concrete repair actions. When the repair actions are more abstract, it results in a larger synthesis space, when repair actions are more concrete, it hampers the likelihood of being able to concentrate on likely repair shapes first. We conjecture that the profile based on CT is better because it enables us to find bigger correct repair shapes (good) in a smaller amount of time (good).

Finally, we think that our results empirically explore some of the foundations of "repairing": there is a difference between prescribing aspirin (it has a high likelihood to contribute to healing, but only partially) and prescribing a specific medicine (one can try many medicines before finding the perfect one).

4.1.3.3 Case Study: Reasoning on GenProg within our Probabilistic Framework

We now aim at showing that our model also enables to reason on Weimer et al.'s example program [12]. This program, shown in Listing 4.1, implements Euclid's greatest common divisor algorithm, but runs in an infinite loop if $a = 0$ and $b > 0$. The fix consists of adding a "return" statement on line 6.

4.1.3.3.1 Probability Distribution In Weimer et al.'s repair approach, the repair model consists of three repair actions: inserting statements, deleting statements, and swapping statements[21]. By statements, they mean AST subtrees. With a uniform probability distribution, the logical time to find the correct shape is 4 (from Equation 4.1). If one favors insertion over deletion and swap, for instance by setting $p_{insert=0.6}$, the median logical time to find the correct repair action becomes 2 which is twice faster. Between 2 and 4, it seems negligible, but for larger repair models, the difference might be counted in days, as we show now.

4.1.3.3.2 Shaping and Synthesis In the GCD program, there are $n_{place} = 13$ places where $n_{ast} = 8$ AST statements can be inserted. In this case, the size synthesis space can be formally approximated: the number of possible insertions is $n_{place} * n_{ast}$; the number of possible deletions is n_{ast}; the number of possible swaps is $(n_{ast})^2$.

This enables us to apply our probabilistic reasoning at the level of concrete fix as follows. We define the concrete repair distribution as: $p_{insert}(ast_i, place_k) = \frac{p_{insert}}{n_{place} * n_{ast}}$, $p_{delete}(ast_j) = \frac{p_{delete}}{n_{ast}}$, $p_{swap}(ast_i, ast_j) = \frac{p_{swap}}{(n_{ast})^2}$.

With a uniform distribution $p_{insert} = p_{delete} = p_{swap} = 1/3$, formula 4.1 yields that the logical time to fix this particular bug (insertion of node #8 at place #3) is 219 attempts (note that it is not anymore a shaping time, but the real number of required runs). However, we

[21] In more recent versions of GenProg, swapping has been replaced by "replacing".

Listing 4.1: The infinite loop bug of Weimer et al.'s bug [12]. Code insertion can be made on 13 places, 8 AST subtrees can be deleted or copied.

```
1     // insert 1
2     if (a == 0) { // ast 1
3       // insert 2
4       System.out.println(b); // ast 2
5       // insert 3
6     }
7     // insert 4
8     while (b != 0) { // infinite loop // ast 3
9       // insert 5
10      if (a > b) { // ast 4
11        // insert 6
12        a = a - b; // ast 5
13        // insert 7
14      } else {
15        // insert 8
16        b = b - a; // ast 6
17        // insert 9
18      }
19      // insert 10
20    }
21    // insert 11
22    System.out.println(a); // ast 7
23    // insert 12
24    return; // ast 8
25    // insert 13
26  }
```

p_{insert}	p_{delete}	p_{swap}	Logical time
.33	.33	.33	219
.39	.28	.33	185
.45	.22	.33	160
.40	.40	.20	180
.50	.30	.20	144
.60	.20	.20	120

Table 4.2: Different probability distributions over the GenProg's repair model.

observed over real bug fixes that $p_{insert} > p_{delete}$ (see Table 3.4 from Section 3.3). What if we distort the uniform distribution over the repair model to favor insertion? Table 4.2 gives the results for arbitrary distributions spanning different kinds of distribution. This table shows that as soon as we favor insertion over deletion of code, the logical time to find the repair does actually decrease.

Interestingly, the same kind of reasoning applies to fault localization. Let's assume that a fault localizer filters out half of the possible places where to modify code (i.e. $n_{place} = 7$). Under the uniform distribution and the space concrete repair space, the logical time to find the fix decreases from 219 to 118 runs.

4.1.3.3.3 Repairability and Fix Size We consider the same model but on larger programs with fault localization, for instance 100 AST nodes and 20 potential places for changes. Let us assume that the concrete fix consists of inserting node #33 at place #13. Under a uniform distribution, the corresponding repair time according to formula 4.1 is $\geq 20{,}000$ runs. Let us assume that the concrete fix consists of two repair actions: inserting node #33 at place #13 and deleting node #12. Under a uniform distribution, the repair time becomes 636,000 runs, a 30-fold increase.

Obviously, for sake of static typing and runtime semantics, the nodes cannot be inserted anywhere, resulting in lower number of runs. However, we think that more than the logical time, what matters is the order of magnitude for the difference between the two scenarios. Our results indicate that it is very hard to find concrete fixes that combine different repair actions.

Let us now be rather speculative. Those simulation results contribute to the debate on whether past results on evolutionary repair are either evolutionary or guided random search [101]. According to our simulation results, it seems that the evolutionary part (combining different repair actions) is indeed extremely challenging. On the other hand, our simulation does not involve fitness functions, it is only guided random search, what we would call "Monte Carlo" repair. A good fitness function might counter-balance the combinatorial explosion of repair actions.

4.1.4 Summary

In this Section we have presented a strategy to optimize the time to navigate the search space. The strategy relies on the probability distribution over repair actions. It focuses on concentrating on likely repair actions first. We present a mathematical analysis to compare

and evaluate probability distributions with the goal of finding that one that optimize the navigation.

We have shown that certain distributions over repair actions can result in an infinite time (in average) to find a repair shape while other fine-tuned distributions enable us to find a repair shape in hundreds of repair attempts.

In the following section we present a second strategy to reduce the synthesis search space of redundancy-based repair approaches such as GenProg.

4.2 Reducing Synthesis Search Space for Software Redundancy-based Repair

In Section 4.1 we presented a strategy to optimize the time to navigate the shaping search space. In this Section we present a strategy that aims at optimizing the repair time for a particular kind of repair approaches: redundancy-based repair approaches.

To some extent, each program repair technique is based on a underlying assumption. GenProg's "secret sauce" [12, 8] is the assumption that large programs contain the seeds of their own repair and thus that re-arrangements of existing statements can fix most bugs. This redundancy assumption is also behind four out of ten PAR's repair templates [5]. We call redundancy-based repair approach to those approaches that work under this redundancy assumption. By contrast, SemFix [10] makes different assumptions: it is based on the idea that some bugs can be repaired by changing only one variable assignment or if conditional expression.

Redundancy-based repair approaches synthesize repairs by picking source code from somewhere in the program. The search spaces of these approaches are formed by source code found in other places of the program. This assumption could produce large search spaces, especially when the program to be repaired is large. By consequence, the navigation of the repair search space could take infinite time (in other words, the approach is not able to find a fix).

In this Section we aim at presenting a strategy to optimize the solution search for redundancy-based repair approaches. Our challenge is to know whether there exist alternative repair search spaces, smaller than the original (defined by one repair approach such as GenProg) and with similar repairability strength. This means, we aim at reducing the size of the space but keeping the number of solutions that the "original" search space has.

This experiment also allows us to validate the redundancy assumption. We wonder whether it makes sense to re-arrange existing code to fix bugs. The results enable us to understand the foundational assumptions of program repair approaches based on redundancy such as GenProg or PAR.

The section remains as follows. In Section 4.2.1 we present two redundancy-based repair approaches from the literature. In Section 4.2.2 we study the search space of this kind of repair approaches. In Section 4.2.3 we present our strategy to optimize the navigation in those spaces. In Section 4.2.4 we present an evaluation of the strategy, then in Section 4.2.5 we present the results of the evaluation. Finally, Section 4.2.6 concludes the section.

4.2.1 Software Redundancy-based Repair Approaches

In this section we present two state-of-the-art program repair approaches: GenProg and PAR. Both approaches work under the assumption that the code of a fix was already written before in the program.

GenProg is an automatic program repair approach guided by genetic programming. It applies evolutionary computation to evolve a failing version, i.e., with one defect, of a program to a version without the defect. It evolves the program applying three kinds of operators: inserting, replacing, and removing source code. The approach relies on the presence of software redundancy: it assumes that the code that forms a fix (the fix ingredients) already exists somewhere in the program. This assumption involves that GenProg always instantiates inserting and replacing operators with code taken from elsewhere in the program. As consequence, it never synthesizes fixes that introduce new code.

Another redundancy-based repair approach is PAR [5]. In contrast to GenProg, the repair operators of PAR correspond to bug fix templates derived from bug fix patterns manually identified. The approach uses existing source code to instantiate four bug fix templates. For example, templates *Expression replace for an if conditional* and *Expression added and removed in if conditional* are instantiated with boolean expression collected in the same scope from the location of the *if condition* defect.

Let us present as example one defect that, in theory, both redundancy-based repair approaches are able to fix. The defect corresponds to an issue reported in Apache Commons Math project [22]. Listing 4.2 presents a chunk of the file that contains the defect. The defect is located in line 16: the *if condition* uses an incorrect relational operator. The fix proposed by developers [23] changes the operator $>=$ by $>$.

GenProg and PAR are able to synthesize this fix (and eventually others that also fix the bug). GenProg could generate it applying replace operator. This operator replaces the AST node that corresponds to the buggy *if condition*, i.e., $fa * fb >= 0.0$, by one AST node of boolean expression, i.e., $fa * fb > 0.0$, taken from line 13 (a *while* statement with a condition formed by three sub-terms). The synthesized patch is equal (same code) to the real fix proposed by the Apache developers.

PAR could also generate this fix by applying *Expression replace for an if conditional* bug fix template. In the same way, PAR replaces the buggy if conditional by the mentioned sub-term found in line 13.

[22] `https://issues.apache.org/jira/browse/MATH-280`

[23] `https://fisheye6.atlassian.com/changelog/commons?cs=791766` fix introduced in file UnivariateRealSolverUtils.java

Listing 4.2: Buggy If condition from issue MATH-280. Redundancy-based approaches are able to fix the bug by replacing its boolean expression by another from the same method.

```
1         double a = initial;
2         double b = initial;
3         double fa;
4         double fb;
5         int numIterations = 0 ;
6
7         do {
8             a = Math.max(a - 1.0, lowerBound);
9             b = Math.min(b + 1.0, upperBound);
10            fa = function.value(a);
11            fb = function.value(b);
12            numIterations++ ;
13        } while ((fa * fb > 0.0) && (numIterations < maximumIterations) &&
14                ((a > lowerBound) || (b < upperBound)));
15
16        if (fa * fb >= 0.0 ) { //buggy if condition; fix:  if (fa * fb > 0.0
17            throw new ConvergenceException(...);
18        }
```

4.2.2 Defining Search Spaces for Redundancy-based Repair Approaches

In Section 4.1.1 we decompose a repair search space into three spaces: *fault localization space*, *shape space* and *synthesis space*. In this section, we focus on the *synthesis space* of redundancy-based approaches. Repair approaches navigate this space to instantiate a repair shape. For instance, GenProg does it to instantiate two of its repair actions: *insert* and *replace*. By the redundancy assumption behind them, redundancy-based approaches never synthesis source code, i.e., they reuse existing code from somewhere in the application. As consequence, their synthesis space is composed of already written code.

In the remaining of the section we focus on how to define a synthesis space. We analyze how the topology of this search space impacts on the performance of redundancy-based repair approaches. For that, we introduce a new measurement of source code redundancy. In the remaining of the section, we define the concept *temporal redundancy* in detail.

4.2.2.1 Fragment Redundancy

We use *fragment* to denote a substring of source code. For instance, the source code line "for (int i=0; i<n; i++)" is a fragment. Fragments are always defined according to a level of granularity.

A fragment F is *snapshot redundant* at time T if another instance of that same fragment F exists elsewhere in the program at time T. This is the redundancy studied by Gabel and Su [36] and used by GenProg [12]. In this work we consider a richer notion of redundancy that includes historical context.

A fragment F is *temporally redundant* at time T if that same fragment F has already been seen during the history of the software under analysis (i.e., at time $T' < T$). For instance,

literal "42" might be added in version #1, be removed in version #2 and reused again in version #3. In the commit of version #3, the "42" fragment is temporally redundant. Thus, once a fragment has appeared it is always subsequently viewed as a potential source of redundancy. We consider the first version of a program to be created by insertions from an empty initial program.

Consequently, a temporally redundant fragment is associated with a birth date, the very first time when it has been used in the software under study. At the point in time it appears, the fragment is called **unique fragment**, and remains unique as long it is not used a second time in a subsequent commit.

4.2.2.2 Fragment Pool

We propose that the repair search space of redundancy-based repair approaches is composed of two components: the search space of fragments (the atomic building blocks) and the search space of their combinations. In this section we focus on the former. We call it *fragment pool*. It has all the fragments seen up in the history of a program to a given point in time. Fragment Temporal Redundancy is measuring using the fragment pool: if a fragment exists in the pool means it is redundant. Otherwise, the fragment is new, i.e., not redundant.

4.2.3 A Strategy to Reduce the Size of the Redundancy-based Synthesis Search Space

In this Section 4.2.2 we present one strategy to optimize the search of the solution in the search space of redundancy-based repair approaches such as GenProg. Synthesis search spaces of this kind of repair approach include source code fragments from the application under repair. However, for large applications, these spaces could contain a large number of fragments. As consequence, the time to navigate them could be large or even infinite. In face to this situation, redundancy-based repair approaches are not able to find a solution to the bug. Remember that repair approaches have criteria to limit the search of a solution in the repair space. For example, the navigation in GenProg is limited by parameters from genetic programming such as the number of generations and the size of the initial population. In one of the most recent experiments [8], these values are 10 and 40, respectively.

Our motivation is redundancy based approaches be able to find undiscovered solutions from the search space. The intuition we have is these undiscovered solutions can be discovered by decreasing the time to navigate the synthesis search space. In this section we present one strategy that aims at finding a solution faster. The strategy aims at defining smaller synthesis spaces without losing *fertility*. That means, to have a search space with a similar number of solutions that the original space has. A smaller search space allows repair approach to transverse all its elements faster than a larger one.

To reduce the search spaces, the strategy defines spaces collecting fragments that are in a given *scope*. According to the selected scope, the space's definition yields different topologies of search spaces. By consequence, this impact on the repair strength of the repair approaches. In this work, the strategy considers two temporal redundancy scopes: *global* and *local*. We present them in the following section.

4.2.3.1 Defining Two Scopes of Temporal Redundancy

A fragment is temporally redundant if that same fragment was written before in the application. As we study redundancy from version control systems, it means a fragment is temporally redundant if that same fragment appeared in a previous commit. This kind of redundancy has a *global scope*: the location of the previous fragment instance does not matter. At global scoping, there is one fragment pool. It contains all the fragments from everywhere in the application.

We now define a more restricted *local scope* notion of temporal redundancy. A fragment is locally temporally redundant if that same fragment has been used in a previous commit to the same *file*.

For example, consider two files, F_1 and F_2, each containing 3 fragments: $F_1 = \{a, b, c\}$, $F_2 = \{d, e, f\}$. Suppose commit C_1 adds fragment c to file F_2. For that commit, fragment c is global temporally redundant (already available in F_1), but not locally temporally redundant (never previously available in F_2). Suppose commit C_2 introduces another version of F_2 replacing fragment e with d. In that commit, fragment d is locally redundant (since d was previously available in F_2).

At local scoping, there is one fragment per each file from the application. A fragment from file F is *locally* redundant if it exists in the fragment pool associated to file F. Otherwise, it is not locally redundant.

4.2.4 Definition of Evaluation Procedure

4.2.4.1 Evaluation Goals

In this Section we aim at evaluating the strategy presented in Section 4.2.3. The strategy reduces the size of synthesis search space by considering fragments included in a given scope. In Section 4.2.3.1 we present two scopes: *local* and *global*. We wonder whether this strategy affects the repair strength of redundancy-based repair approach. Our intuition is local scope allows repair approach to define smaller search space without losing repair strength. We set up an evaluation to validate these ideas.

The evaluation analyzes the software history of applications. We aim at measuring the temporal redundancy of commits. We carry out the experiment considering global and local redundancy levels. Our goal is to prove that: a) the assumption behind redundancy-based approaches makes sense, i.e., it has sense to reuse already written code to synthesize commits; and b) the strategy of reducing the search space allows repair approaches to synthesize commits using a smaller search space than without the strategy.

4.2.4.2 A Method to Validating Redundancy-based Assumption

In this section we present an experiment to validate the redundancy assumption used by redundancy-based repair approaches such as GenProg. We ask whether it makes sense to rearrange existing code or code changes to fix bugs. For that, the experiment aims at studying how the software evolves, in particular whether the code that is added to the application in the evolution was already written before in the application.

In our experiment we analyze version control systems (VCS). We analyze the source code introduced by each commit from the VCS . We want to measure the number of commits that introduce only already written code i.e., *redundant fragments*. This measure allows us to

validate the redundancy assumption: In theory, a redundancy-based repair approach should be capable of synthesizing the code of these commits.

As we analyze commits from a version control system, in the following section we define a redundancy measure at the commit level.

4.2.4.2.1 Defining Commit Redundancy A *commit* in a version control system consists of new file versions. Conceptually, a commit can be viewed in two ways: as a set of file pairs (before and after the commit), or as a set of changes (applied to the version before the commit to obtain the version after it).

In this section, we use this change-based view of commits. We consider that a commit adds and/or deletes new source code fragments. An *update* is considered as the combination of a deletion and an addition. We do not consider commits done on other artifacts than source code.

A source code commit is composed of added and/or deleted fragments. Using a cooking metaphor, the added fragments are the "ingredients" of the commit.

We define a *temporally redundant commit* as a commit for which all added fragments are (individually) temporally redundant. More formally, we define a commit C_j performed at the time T_j as a set S_j of added fragments and a set R_j of removed fragments. Let C be the set of all commits for a program. Then a temporally redundant commit C_j satisfies

$$\forall f \in S_j | \, \exists C_i \in C | \, T_i < T_j \wedge f \in S_i$$

For such commits, no new fragments are invented and no fresh material is introduced: the commit is only a re-arrangement of insertions that have already been seen in previous commits.

Along the same line, a unique commit only introduces unique fragments and a partially redundant commit introduces already written fragment as well as unique fragments. Along the same line, a partially redundant commit C_j satisfies $\exists e_m \in S_j, \exists C_i | \ldots$ and the formal definition of **unique** commits is trivial.

4.2.4.2.2 Commit Classification Example Let us present an example of commit classification at line-level granularity. In the example, the software history of a program contains four commits $C_i, i = 1..4$. A commit is represented by a set of fragments F_x and a date d_i where the commit was done. Being the fragment history:

$C_1 = \{F_a, F_b, F_c\}, C_2 = \{F_b, F_d\}, C_3 = \{F_e\}, C_4 = \{F_c, F_d\}$ and $d_1 < d_2 < d_3 < d_4$.

Let us classify the commits:
C_2 is *partially redundant*. It introduces existing code (F_b from C_1) but also unknown code (F_d).
C_4 is *redundant*. It introduces exclusively existing code (F_c from C_1 and F_d from C_2).
On contrary, C_1 and C_3 are *unique*, they introduce unknown source code at time d_1 and d_3, respectively.

4.2.4.3 Experimental Protocol

Given a level of granularity and a scoping level, our experimental protocol to measure the temporal redundancy of the evolution of a program consists of the following phases: It is depicted in Figure 4.4.

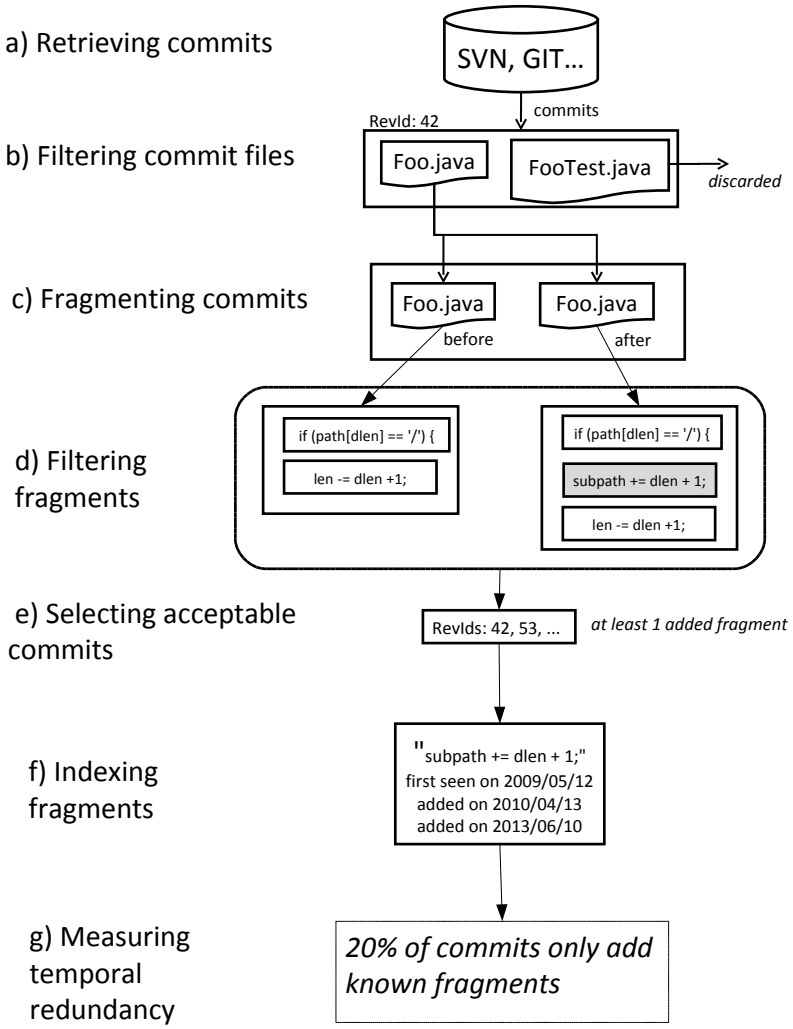

Figure 4.4: OVERVIEW AND EXAMPLE OF OUR METHODOLOGY FOR CALCULATING SOFTWARE TEMPORAL REDUNDANCY (LINE LEVEL GRANULARITY)

a) Retrieving commits. All commits of the program under analysis are collected from the repository.

b) Filtering commit files. Only commits to executable code are retained; commits to test cases are discarded.

c) Fragmenting commits. We split each relevant file into fragments at a given level of granularity (e.g., lines or tokens, see Section 4.2.4.4). This results in a before-commit and an after-commit sequence of fragments. At line and token granularity level, we use the Myers differencing algorithm [14] to compare both fragment sequences and obtain the added fragments of the commit.

d) Filtering fragments. We filter out whitespace and comments. In this paper we are only interested in the evolution of executable code and not in indentation or documentation.

e) Selecting acceptable commits. We select those commits that introduce at least one fragment after filtering. We call such commits *acceptable*.

f) Indexing fragments. We consider each added fragment in each acceptable commit in ascending temporal order. If a fragment has not been encountered previously at the given scoping level (i.e., global or local), we index it with the date of its first introduction. For a global scope level, we define one fragment pool where we store all fragment introduced by commits. For a local scope level, we define one fragment pool per each *file* of the application. Each of those pools contains the fragments written in the history of the associated file.

g) Measuring temporal redundancy. The *temporal redundancy* of the entire program's evolution is the fraction of acceptable commits that are temporally redundant (see Section 4.2.4.2.1).

4.2.4.4 Fragment Granularities

We consider two different levels of granularities of source code fragments: *Lines* and *Tokens*.

At *line* level, we split a source code chunk in lines, as separated by line breaks. For example, for the following source code chunk:

```
1 int i = getSum();
2 if ( j > 1000){
```

We obtain two fragments at the line level, one per each line: $int\ i\ =\ getSum()$; and $if(j > 1000)$.

At the *token* level, we split one source code chunk in tokens, as separated by lexing rules. From the former listing, we obtain seven fragments: int, i, $getSum()$, if, j, $>$ and 1000.

4.2.5 Empirical Results

We now present our empirical results on the temporal redundancy of software (as defined in Section 4.2.4.2.1) following the experimental design presented in Section 4.2.4.3. First, in subsection 4.2.5.1 we present the project used in the evaluation. Then, in subsection 4.2.5.2, we present the result of the validation of the software redundancy assumption. Finally, in subsection 4.2.5.3, we analyze the result of the space size reducing strategy.

4.2.5.1 Dataset

We use six open-source Java projects to measure the temporal redundancy. They are: Apache Log4j, JUnit, Picocontainer, Apache Commons Collections, Apache Commons Math, and

Table 4.3: The temporal redundancy of six open-source applications.

Program	Acceptable Commits	Line granularity				Token granularity			
		Global		Local		Global		Local	
		Temporal redundancy	Pool Size	Temporal redundancy	Pool Size	Temporal redundancy	Pool Size	Temporal redundancy	Pool Size
C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
log4	1687	**9%**	43313	**6%**	57	**39%**	14294	**19%**	71
junit	713	**17%**	8855	**16%**	18	**43%**	3256	**29%**	72.5
pico	157	**3%**	16911	**2%**	22.5	**31%**	6273	**8%**	46
collections	1019	**7%**	25406	**4%**	35	**52%**	4163	**23%**	85.5
math	2210	**6%**	69943	**4%**	37	**45%**	20742	**18%**	100.5
lang	1290	**8%**	22330	**6%**	63	**50%**	6692	**29%**	98

Apache Commons Lang. The inclusion criterion is as follows: the Apache projects were used in previous research on automatic program repair [5], while the remaining two are Java projects mentioned in previous research on software evolution [102]. After applying the filters presented above on the 16071 commits of the dataset, we obtain 7076 acceptable commits.

4.2.5.2 Measuring Temporal Redundancy

4.2.5.2.1 Line-Level Temporal Redundancy *Research question: What is the amount of line-based temporal redundancy?*

For each application of our dataset, we measured the total number of acceptable commits within the analysis timespan (the complete data is available at `http://goo.gl/k0rZWc`) and the global-scope line-level temporal redundancy. Columns #2 and #3 of Table 4.3 report the results.

For instance, for log4j (the first row), there are 1687 commits which add at least one executable line. Only 9% of those 1681 are temporally redundant commits.

Overall, at the level of lines, 3–17% of the accepted commits are temporally redundant commits. Their basic ingredients are only previously-inserted code.

This has additional implications for automatic repair and code synthesis: for synthesizing those commits, the search-space has a finite number of atomic building blocks (previously-observed line-level fragments). Theoretically, a redundancy-based approach should be able to synthesize all the temporally-redundant commits.

The interval 3–17% is large and the reasons behind this variation are not obvious. One reason could be that the commit conventions used by the developers of a project are different. For instance, some projects prefer to have small and atomic commits (one bug fix or feature per commit). Other projects are less restrictive on this point. This has a direct impact on the redundancy: small and atomic commits are more likely to be redundant.

4.2.5.2.2 Token-Level Temporal Redundancy *Research question: Is there a difference between line-based and token-based temporal redundancy?*

Before answering this question, we note that, analytically, all temporally redundant commits at the level of lines are necessarily temporally redundant commits at the level of tokens.

Furthermore, a unique new line might be exclusively composed of existing tokens. Consequently, the token-based temporal redundancy must be equal to or greater than line-based temporal redundancy. We now measure the temporal redundancy at the line and token level.

Table 4.3 reports global scope line-level (column #3) and token-level temporal redundancy (column #7). For instance, in log4j, there is a line-level temporal redundancy of 9% but a token-level redundancy of 39%. Overall, at the token level, 29–52% of commits are temporally redundant.

For all projects, token-based temporal redundancy exceeds line-based. This follows from the analytical argument above and gives confidence in the experiment's construct validity.

Overall, token-level temporal redundancy (between 29% and 52%) is much higher than the line-level temporal redundancy. For automated repair and code synthesis, a high temporal redundancy implies a smaller search space. This holds for both the line and token level of granularity. Our token-level temporal redundancy measurements imply that for between 29% and 52% of accepted commits, synthesis and repair need never invent a new token. For instance, repair or synthesis of arithmetic code need only consider recombining existing literals and operations for one-third to one-half of commits.

4.2.5.3 Redundancy Scope Experiment

The results presented in Section 4.2.5.2 allow us to validate the redundancy assumption behind redundancy-based repair approaches. We can affirm that it makes sense to reuse to synthesis commits from version control systems. In this section we focus on validating the strategy to optimize the navigation of the search space presented in Section 4.2.3. For that we compare two measures for local and global scopes: amount of temporal redundancy and size of the fragment pool.

Let us first analyze the differences fragment pools for the same scope, in particular, global scope. Table 4.3 gives the size of the global scope fragment pool for line-level (column #4) and token-level (column #8) analyses. For instance, in the considered slice of history of log4j, there are 43313 different lines (i.e., size of global fragment pool) and 14294 different tokens that are involved in the software evolution. For all applications under study, the token pool at the point in time of the last commit is much smaller than the line pool.

For automated repair or code synthesis, there is a tension between working with the line pool or the token pool. To some extent, the temporally redundant commits correspond to the number of commits that can be synthesized. With the line pool, the combination of lines is much smaller (the combination space is smaller) but fewer commits can be synthesized ($\sim$10%). With the token pool, more commits can be synthesized ($\sim$40%), but at the price of exploring a much bigger combination space.

Now, let us focus on evaluating the strategy. Our research question is: *Do local (that is, file) scope restrictions impact the amount of temporal redundancy?*

We now measure the temporal redundancy available at the local scope in the same file (as defined in Section 4.2.3.1).

Table 4.3 reports global and local scope temporal redundancy in our dataset. For each granularity, there is one column "Global" and one column "Local" corresponding to the different scope. For instance, at the line level, column #3 is the temporal redundancy at the global scope and column #5 gives it when considering a local scope.

As discussed in Section 4.2.5.2.1, at the line granularity, there are between 3% and 17% of temporally redundant commits at the global scope. At the local scope, there are between 2%

and 16%.

The temporal redundancy of both scopes is of the same order of magnitude. In all projects, more than half of the temporally redundant commits actually have local temporal redundancy. Consequently, *at line granularity, most of the temporal redundancy is localized in the same file*.

At token-level granularity, the results are similar: we find a large amount of token-level local-scope temporal redundancy. Tokens are likely to be reused in commits impacting the same file. This further indicates that the fragment locality matters during software evolution. We note that the difference of redundancy between global and local scopes is slightly higher at the token level (col. #7 vs. #9) than at the line level (col. #3 vs. #5).

These results validate our strategy for reducing the repair time presented in Section 4.2.3. We observe two main conclusions. First, at the line level, the local scope pool is able to seed the same order of magnitude of commits as the global one. In other words, it is almost as fertile as the global pool. Second, when one considers the local scope pool, the search space is much smaller. For instance, for log4j, the median local pool size at the line level is 57 lines, compared to 43313 at the global scope level. As consequence, the time to navigate a local pool is much smaller than the time to navigate the global pool.

Restricting attention to the local scope reduces the search space greatly while still enabling the synthesis of a large number of commits. Those results are directly actionable for improving GenProg and other redundancy-based approaches: our results indicate that applying a strategy to reduce the search space by only considering local redundancy would decrease the repair time while keeping a high repair success potential.

4.2.6 Summary

In this section we presented a strategy to reduce the time to navigate a synthesis search space for redundancy-based repair approaches such as GenProg. The strategy proposes to reduce the synthesis search space, composed by already written source code from the application to repair. A reduced space contains code that belongs to a given scope. In the evaluation we consider two scopes: *local* (file) and *global* (all application). We have shown that considering a local scope allows repair approaches to have smaller space without resigning repair strength. This evaluation also allows us to validate the assumption behind redundancy-based repair approaches such as GenProg. It makes sense to use already written source in the application to synthesize bug fixes.

4.3 Conclusion

In this chapter we presented two strategies to optimize search-based repair approaches. The first one helps to reduce the time to find a solution, i.e., to navigate the search space. The strategy relies on source change probabilities taken from version control system. It first selects those frequent repair changes. We proved that there are distribution probabilities over repair models that are better than others, i.e., it allows approaches to find faster a solution. The second strategy reduces the search space from redundancy-based repair approaches such as GenProg [12]. We proved that these kinds of approaches can reuse source code from the file where the bug is located to synthesis fixes.

Chapter 5

A Unified Repair Framework for Unbiased Evaluation of Repair Approaches

In this thesis one of the main motivations is to improve the repairability of bugs. This improvement means to increase the number of bugs fixed by repair approaches. Repair approach evaluations from literature show that a fraction of these defects remain unrepairable. For instance, GenProg is able to repair 55 out of 105 defects, remaining 50 unrepairable [8]. In Section 4 we have presented two strategies that help repair approaches to reduce the time of searching bug fixes.

We observe that the proportion of repairable and unrepairable defects depends on the way the evaluation dataset is built. A dataset could include defects that an approach is able, at least in theory, to repair, and defects that are not repairable by construction. If one defines a dataset that includes majority of repairable defects, the repair efficacy of the approach should be higher. Otherwise, if the majority of the defects are unrepairable by the approach under evaluation, the repair efficacy of the approach should be low. As consequence, the way the dataset is built can bias the result of the evaluation.

In this chapter we propose a method to obtain conclusive evaluations of automatic software repairs. For that, we need to characterize how the dataset is built and what it contains. In Section 5.1, we first present a methodology to define evaluation datasets with a controlled bias in the dataset definition. This dataset contains defects of a unique defect class.

Then, we focus on the execution of a repair approach evaluation. Our motivation is to execute evaluations of repair approaches that produce reliable and conclusive results. We aim at setting up an experiment that allows us measuring the real strength of the repair operators used by the evaluated approaches. This experiment allows us to instantiate our methodology of repair approach evaluation in order to validate it. In particular we aim at studying the repairability of a particular defect class: *if conditions*. Those defects are common: previous works [9, 11] have shown that there are the most repaired elements in source code. For instance, the results of Pan et al.[9] over six open source projects, between 5% and 18.6% of bug fix commits are modifications done in *if conditions*.

In our study, we consider three repair techniques from the most authoritative literature: GenProg [12], PAR [5] and the mutation-based approach defined by Debroy and Wong [13].

We propose a framework that unifies differences across the repair approaches, and encodes the particular repair operators of them.

In this chapter we propose: in Section 5.1, we first present a methodology to define evaluation dataset and a dataset of *if condition* defects. In Section 5.2 we present a methodology to develop repair approach evaluations. For subsequent work, we explain the decision taken to replicate the approaches. Finally, in Section 5.3 we present the results of the evaluation of the three repair approaches.

5.1 Defining Defect Datasets for Evaluating Repair Approaches

In this section, we design a procedure to evaluate repair approaches. In particular, we focus on the design of the evaluation dataset due to a main reason: we need unbiased evaluations. We present a procedure to evaluate repair approaches with controlled biases.

5.1.1 Defining a Defect Class

A *defect class* is a family of defects that have something in common [103]. There are three dimensions for defining a defect class: *a*) the root cause e.g., the use of a not initialized variable, *b*) the symptom e.g., a *null pointer error exception*, and *c*) the kind of fix e.g., initialization of a variable or addition of a null-pointer checker precondition.

A repair approach should always target explicit defect classes [103]. These defect classes could be repaired by an approach. For some approaches such as Semfix [10] the target defect classes are specific and explicit: it fixes *a*) *if condition* defects and *b*) *integer initialization* defects. On the contrary, approaches such as GenProg [12] do not explicitly target any defect class. However, more recent GenProg's evaluation [1] shows that the approach fixed *segfault*, *infinite loop* and *buffer exploit* defects.

5.1.2 Bias in Evaluation Datasets

A typical method used in previous publications [12, 5, 77, 13] to evaluate a repair approach consists of taking, one by one, defects from a defect dataset and trying to find a solution, i.e., a repair, using the repair approach. As discussed in [103], the way one builds a dataset directly impacts on the evaluation result. For example, let us consider an approach $appr_1$ that targets defect classes A and B, and another approach $appr_2$ that targets classes B and C. The evaluation of those approaches using a defect dataset formed with 80% class A and 20% of class B would clearly favor $appr_1$. Biased evaluation result can arise when the dataset is built without following defined inclusion criteria.

We claim that the procedures for defining an evaluation dataset and for evaluating a repair approach should be as separated as possible. A dataset tailored for a particular approach evaluation could produce inconclusive results. Both procedures should share only one single concept: *a defect class*. This means the dataset should contain only defects from the same defect class; and the repair approach targets defects from that class. Then, the dataset can be considered as well formed with respect to this defect class, and competing approaches can be quantitatively compared. From the previous example, a conclusive evaluation could be that one that defines, for instance, three defect datasets, one for each target defect: dataset with defect class A, dataset with defect class B, and finally dataset with defect class C. Now,

the evaluation can concentrate on evaluating the repairability over one each class: $appr_1$ fixes more defects of class A that $appr_2$, but the latter fixes more defects from class C.

In this section we present a methodology to built evaluation datasets with a specific focus on minimizing the evaluation biases and fallacies. Moreover, we illustrate it to build a dataset of *if condition defects* for repair approach evaluations.

5.1.3 A Methodology to Define Defect Datasets

The challenge we tackle is to define a methodology to build defect datasets for evaluation of automatic software repair aproaches. The methodology must fulfill the following requirements: *a*) the dataset includes inclusion criteria of defects, such as the target defect class; *b*) for each included defect, the source code of the defective version is publicly available for evaluation replication; *c*) the defects included are reproducible; *d*) optionally, it includes defects reported in, for example, bug trackers or mail lists; these sources of information allow us to better understand, for instance, the defect's causes, their proposed repairs and their priority.

We propose a methodology that has two inclusion criteria: one that defines criteria for project selection (Section 5.1.3.1), and another that defines criteria for selecting defects (Section 5.1.3.2).

5.1.3.1 Methodology for Choosing Projects

In this subsection we present a methodology to select a project to search defects. We enumerate the most important criteria that, for us, should be considered in the project selection.

5.1.3.1.1 Availability of project history Collecting defects from a project involves finding versions that contain one or more defects. For that, it is necessary that a project has publicly available either: *a*) a set of isolated versions of a program (e.g., v.1.1, v.1.2); or *b*) a version control system (VCS) such as CVS, GIT or SVN that manages versions of a project through its development and maintenance life-cycle.

5.1.3.1.2 Project features Larger projects, in terms of number of revisions and versions, increase the probability to have a richer history and, by consequence, more defects to include in the dataset.

5.1.3.1.3 Existence of correctness validation mechanism Each version of the project must include, at least, one mechanism to automatically measure and validate its correctness according to the program specification. Moreover, these mechanisms must cover at least the most critical components. An example of a validation mechanism is a *test suite*. Repair approaches such as GenProg, PAR and SemFix [10] use test suites as a mechanism of validation of their repairs.

5.1.3.1.4 Availability of reporting track system A project with publicly available issue tracking system or mailing lists allows us to collect defects from there. For example, we will able to query tracking systems to collect those issues labeled as "bug".

5.1.3.1.5 Conventions, rules and best practices Coding rules or management of VCS conventions used in projects help to increase the software quality. We focus on three conventions: *1*) the use of atomic commits to introduce new features or bug fixes; *2*) the inclusion of a description in the commit message log of the changes introduced by the commit; *3*) the inclusion of a link to the issue tracker in the message log when changes correspond to a reported issue.

These three conventions will help us to filter commits that introduce bug fixes from the version control systems in an accurate way.

5.1.3.1.6 Summary In this subsection we presented a methodology to select software projects to be used in evaluations of software repair approaches. In the following subsection we present a methodology to collect defects from those projects.

5.1.3.2 Methodologies of Collecting Defects

The methodology collects defects of one target defect class such as *if condition defects*. A technique for collecting defects, used for example in [89], first searches commits that introduce fixes; then, it searches for the versions just before those commits (without the fix changes) that probably contain defects.

There are two methods to collect defects. One is to first navigate reports from issue tracking system or a mailing list of a project to find defects, and then to obtain the versions with those defects. The other way is to first collect from the VCS commits that include defects, then, to analyze the associated issue reports. For that, it is necessary to link VCS commits that fix bugs with the issue reports of those bugs. Proposed techniques such as the one presented by Fischer et al. [46] can be used to automatically discover these links. Once a commit is linked to one issue report, a validation step determines whether the linked issue is a bug or not. Antoniol et al. [48] study reports from issue tracker and obtain that less than half of them were related to corrective maintenance (bug fixing). A risk of bias could be present in the linkage heuristic used. For instance, Bird et al. [47] found that heuristics based on mining explicit links could produce bias results. A reason is that developers can omit bug references (the links) in the commit message log. Approaches such as Wu et al. [49] have emerged to discover missing links. In case these approaches can be used to link and collect more defects, the criteria behind the linking process should be clearly specified and included in the dataset definition. In the context of automatic software repair, this bias could affect the evaluation of the repairability of a given defect class. The repairability of reported defects could be different (easier or harder) from those that were not reported in the issue tracker.

Both methods for collecting defects are equally valid. In the following section we implement the second method. We first collect VCS commits and then we analyze the issue reports of linked bug fix commits.

5.1.4 Methodology Implementation

In this section we present an implementation of the methodology to build a sound dataset. In the presented implementation we focus on collecting *if condition* defects.

The implementation has three steps:

1. Automatic filtering of *if condition* fixing commits;

2. Manual validation of the commit content; and

3. Validation of defect reproducibility.

The implementation combines an automatic and a manual processing. It starts by automatically mining defects from version control systems (VCS) (step 1). The advantage of this processing is that it allows the automation of candidate defects search. Then, a manual processing validates those candidate defects to remove noisy results (step 2) and validates whether they are reproducible (step 3).

The implementation of this method defines defect dataset destined for evaluations of repair approaches known as *test suite-based program repair*. This kind of repair approach uses Unit test for validating the repair correctness. A program satisfies its specification if the test suite passes all the test cases. Otherwise, the program contains a bug. Examples of those kind approaches are GenProg [12], PAR [5], SemFix [10], PATCHIKA [2], AE [78], Debroy and Wong [13].

5.1.4.1 Automatic Filtering of *If Condition* Fixing Commits

The process is guided by one defect class. A defect class groups a family of defects that have something in common [103]. In this work, we consider that defects of a particular class share the same *kinds of fixes*. For example, a defect class groups defects fixed by modifying assignments, another class groups those fixed by modifying *if condition* statements. In this work we analyze the latter defect class.

The automatic process starts by selecting those commits that introduce:

1. at least one source code change to fix an *if condition* defect;

2. code in test cases files to validate the introduced fix, for example, through JUnit assertions; and

3. a link in the commit message log to the defect issue from the issue tracker.

For the first requirement, we use the abstract syntax three (AST) based technique presented in Section 3.4 to mine instances of change patterns in commits. The technique encodes change patterns using a taxonomy of changes over an AST. To mine *if condition* defects, we encode one change pattern: *Update of if condition*. The process mines instances of this pattern in commits and it saves those commits that have at least one instance. In case one wants to define a dataset of another defect class, it is necessary to encode the change pattern related to that defect class. For example, to define a dataset of missing precondition for evaluate repair approaches such as Nopol [104], we can encode a change pattern *Addition of If condition* as we have done in Section 3.4.5.3.

For the second requirement, the process checks if one commit introduces modifications in one or more test case files. The process selects those files which name finishes in "Test.java". For the last requirement, the process searches for explicit issue report links in the commit message log. For instance, to match those explicit links in Apache Commons Math project, we use the following regular expression:

$$(MATH|Math|math) - [0 - 9]+$$

If all those requirements are fulfilled for one commit, we call it "candidate commit" and we consider it for a subsequence manual validation.

5.1.4.2 Manual Validation of Commits Content

Through a manual processing, a candidate commit is evaluated to determine whether it introduces a fix or not. For example, it could introduce an improvement or a new feature. We propose two validations. One is the analysis of linked issue report: we keep those commits that are related to bug reports. The other is a validation at the source code level. The commit diff can introduce:

1. one update of *if condition*; it corresponds to the fix.

2. one update of *if condition* (the fix) plus other changes not related to bug fixing such as refactor changes.

3. *if condition* updates plus other changes where all of them are part of the same fix (We call it a complex fix).

We keep those commits that only introduce fixes for *if condition* defects, that is, the first two cases listed above.

5.1.4.3 Validation of Defect Reproducibility

Finally, we process each fix commit that passes the manual validation. First, we retrieve the previous version to the fix commit. That version contains the defect. Then, we verify whether the defect is reproducible. This involves executing the retrieved version (e.g., through test suite) and to observe whether the defect is exposed (e.g., failing test cases).

5.1.5 Dataset of *If Condition* fixing Defects

In this section we present the dataset of *if condition* defect class using the methodology previously presented. The dataset is publicly available at `http://goo.gl/AS1Yj9`.

5.1.5.1 Target Projects

The dataset is formed by defects of Apache Commons Math[24] and Apache Commons Lang[25]. We select both since: *a*) they have a large history: for Math project more that 4700 commits in 10 years of development; for Lang project, more that 3700 commits in 11 years; *b*) they contain publicly issue trackers; *c*) the developers link commits with reports from the issue tracker: Apache Commons Math and Lang use keywords "MATH-*n*" and "LANG-*n*", respectively, where n is the issue number; and *d*) previous repair approaches such as PAR include defects from these projects in their evaluation. We analyze the commits from the 5/12/2003 to 7/08/2013 for Math and from 19/07/2002 to 7/08/2013 for Lang.

Project	Type-Priority	Issue Tracker Identifier
MATH	Bug-Minor	238, 240, 243, 309, 644, 691, 722, 836
	Bug-Major	198, 273, 280, 288, 340, 780, 904
	Bug-Critical	947
LANG	Bug-Minor	428
	Bug-Major	719, 746

Table 5.1: Overview of our dataset of *if condition* defects. The bugs come from the Apache Commons Math and Apache Commons Lang projects. The dataset has been carefully crafted to minimize the biases.

5.1.5.2 Resulting Dataset

Table 5.1 shows the 19 *if condition* defects that form the dataset, grouped by the priority (minor, major, critical) specified in the associated bug reports. For Math project the automatic process returns 41 candidates *if condition* fix commits. 28/41 were linked to their corresponding reports from the issue tracker and validated as bugs. We discarded 12/41 defects. Those were related to: improvement or new feature issues (4 of them), complex fixes i.e. bug located in more than one statement (5), or not reproducible (3). We accept 16/41 defects. For Lang project, the automatic process returns 18 candidate *if condition* fix commits. All of them were related to issue reports. We accept 3/18 defects, the remaining were changes related to improvement issues (4), complex fixes (9) and not reproducible bugs (2). Regarding with the critically of the defects, our dataset contains the same number of major and minor defects (9 issues each category).

5.1.5.3 Advantages of Our *If Condition* Defect Dataset

Previous works have defined datasets for evaluating their repair approaches [12, 5, 2]. For instance, the authors of PAR approach [5] define a dataset with 119 defects. If we compare the dataset sizes, our dataset is smaller: it has 19 defects. However, in our opinion, our defect dataset is more meaningful for evaluating automatic software repair approaches. Let us explain why. As we present in this Section, our dataset is built with a clear definition: It is a collection of defect of one particular defect class (*if condition* defects), exposed through unit test execution. Contrary, PAR's authors do not include a dataset build criterion. It is not possible to determine neither: a) the defect classes of the included defect (and by consequence, the abundance of each defect classes); b) the criteria for justifying the inclusion of defects; nor c) the criteria for justifying the absence of defects.

On the contrary, using a well-defined dataset is possible to measure: a) the defect class that a repair is able to fix; b) the proportion of repairable defects from a defect class; and c) the defect class that the approach is not able to repair (i.e., the defect class of the unrepaired defects from the dataset).

In our opinion, a clear-defined dataset but smaller than another without a clear definition criterion, gives more conclusive results about a repair approach evaluation. For this reason,

[24]http://commons.apache.org/proper/commons-math/
[25]http://commons.apache.org/proper/commons-lang/

we believe our dataset can be used for meaningful repair approaches evaluations.

5.1.5.4 Summary

In this section we presented a methodology to define defect dataset for automatic software repair evaluations. Using the methodology, we defined a dataset with defects of the same defect class: *if condition* defects. The dataset can be used to evaluate existing or future repair approaches that target the *if condition* defects, making easier the comparison of evaluation results. This dataset has the advantage that its definition is not influenced by any decision that could favor a particular repair approach. Moreover, it does not analyze what kinds of fine-grained changes are involved in an *if condition* update. For example, an *if condition* update could correspond to a relational operator change (the mutation-based approach targets it) or to the addition of a new term in the *if expression* (PAR [5] targets it). This decision avoids having bias to a particular kind of bug fix change in *if condition* fixing and, by consequence, it avoids favoring a particular repair approach. Finally, we implemented the methodology to build datasets that target other defect classes. For example, we could define a dataset of missing method invocation defects to evaluate PATCHIKA [2].

5.2 A Repair Framework for Fixing *If Condition* Defects

In section 5.1 we present a methodology to define evaluation dataset. Now, in this section we aim at presenting a method to evaluate the performance of repair approaches. The combination of both methodologies allows us to design automatic repair approach evaluations with controlled bias. In particular, we focus on the repairability of one particular defect class: *if condition* defects. Empirical studies over software repositories [9, 11] show that changes in *if condition* are some of the most frequent changes done by developers to repair defects. Our motivation is to know whether major and recent automatic repair techniques are able to synthesize fixes for these defects. For this purpose, in Section 5.2.1 we first introduce the *if condition* defect class. Then, we analyze state-of-the-art repair approaches that, in theory, target to *if condition* defect class. In Section 5.2.2 we present a unified repair framework used to replicate those approaches for measuring the repairability of *if condition* defects.

5.2.1 Repair Approaches that Target *If Condition* Defects

The goal of this section is to determine whether three repair approaches from the literature are able, at least in theory, to repair *if condition* defects. In this section, we first introduce the *if condition* defect class (Section 5.2.1.1). Then, we present a typical design of search-based repair approaches, shared by the repair approaches under consideration (Section 5.2.1.2). Finally, we justify why the selected repair approaches target *if condition* defects (Section 5.2.1.3).

5.2.1.1 Defining *If Condition* Defect Class

The *if conditions defect class* characterizes those defects that can be fixed *modifying* an *if condition* expression. This class encompasses a diversity of kinds for source code changes. For example, the update of one relational operator for $>$ to $>=$; the addition of a new term with a logical operator for $if(a > b)$ to $if((a > b)\ \&\&\ (c < f))$; or the removal of one arithmetic operator and one constant for $if(a\ >\ (b + c + d))$ to $if(a\ >\ (b + c))$.

5.2.1.2 Design of Repair Approaches

The search of valid repairs is a kind of search-based software engineering. The repair search space is a set of candidate program fixes. To find a solution, i.e., a repair, an approach navigates the repair search space. The navigation involves selecting one candidate repair and then to determine whether it is valid or not. Weimer et al. call this kind of repair approach design *Generate and Validate* [78].

From previous approaches we identify a repair space as the product of two spaces. One is the *fault localization space*. It contains those source code elements (classes, methods, statements, etc.) that are suspicious to contain a bug. The other, the *fix synthesis space*, contains all possible candidate repairs for a given suspicious statement. Each of those spaces can be navigated on different way, for example, randomly or in a defined order.

Once a candidate repair is selected from the repair space, repair approaches determine whether a candidate repair is valid or not. For this, repair approaches need automatic correctness oracles, which automatically verify whether a program is valid with respect to its specification. The specification defines the target behavior of the program. GenProg uses test suite as oracle of program correctness, i.e. as a proxy to the program specification. We call this kind of approaches *test suite-based program repair*. If the repaired program passes all test cases from the test suite, it means the program satisfies the program specification encoded in the test cases.

5.2.1.3 Analyzing State-of-the-Art Repair Approaches

In the previous subsection, we present the basic design of repair approaches that follows two paradigms: *Generate and validate* and *Test suite-based program repair*. In this section we select three major and recent automatic repair techniques from the literature: GenProg [12], PAR [5] and the mutation-based repair approach from Debroy and Wong [13]. The three of them follow the mentioned two paradigms. In the remaining of this thesis, we use these approaches to carry out our experiments. As we aim at analyzing the repairability of *if condition* defects, in this subsection we analyze whether they are able to repair this defect class.

5.2.1.3.1 GenProg *GenProg* applies evolutionary computation to evolve a failing version of a program, i.e., with one defect, to a version without the defect. It evolves the program applying three kinds of operators: inserting, replacing, and removing source code. The approach relies on the presence of software redundancy: it assumes that the fix probably exists somewhere in the program. For that reason, GenProg takes the code to insert from elsewhere in the application, and never synthesizes fixes that introduce new code.

As GenProg does not explicitly target to defect classes, it is not possible to predict in advance whether GenProg can fix a given defect class or not. Let us suppose a defect is in conditional statement $if(a > b)$, and it can be repaired by changing the relational operator for $>$ to $>=$. GenProg could synthesize this repair if a boolean expression $(a >= b)$ exists somewhere in the program. As consequence, it is not possible to predict the repairability of this defect without analyze the remaining code of the program. The operators defined by GenProg give us the suspicion that it is capable of repairing *if condition* statements. For example, GenProg can create a candidate repair by replacing one suspicious if expression (or one of its sub-terms) by a term taken from another *if condition* located in the program.

From an analytical point of view, GenProg is able to fix some *if condition* defects, but not all of them.

5.2.1.3.2 PAR The second approach analyzed in this work is *PAR* [5]. In contrast to GenProg, the repair operators of PAR correspond to bug fix templates derived from bug fix patterns. We found two of ten templates are able to fix *if condition* defect. They are: *1*) Expression replace for an *if condition*; *2*) Expression added and removed: it inserts or removes a term of the if predicate. For example, the *Expression remover* template modifies a *if condition* expression for $if(a > b \;\&\&\; c < d)$ to $if(a > b)$. PAR also relies on the software redundancy assumption: the expression involved in addition and replacement operators are collected in the same scope of the *if condition* defect location.

5.2.1.3.3 Debroy et al.'s mutation-based repair approach The last repair approach is proposed by Debroy and Wong [13, 105], and is derived from the mutation testing field. For a buggy version of one program the approach generates mutants of that program by applying mutation operators. They apply mutation operators defined in the literature of mutation testing. They define eight categories of mutation operators, five of them can be applied in *if condition* expressions. These operators involve the replacement of: *a*) Op1: basic arithmetic operator for another of the same class; *b*) Op2: Relational operator; *c*) Op3: Logical operator; *d*) Op4: boolean value; *e*) Op8: Decision negation. The approach is not based on evolutionary computation as GenProg or PAR. All first-order mutants (that is, program with exact one operator applied) are generated at once. Then, the approach evaluates them (all or a sub-set of them) one by one to find a valid mutants.

5.2.1.4 Summary

In this section we presented a study of the repair operations with three approaches that all of them are able, at least in theory, to synthesize repairs for *if condition defects*. In the remaining of this paper we set up an experiment to verify whether those approaches are able to repair real defects from this class. In the following Section we present a repair framework to replicate those approaches.

5.2.2 A Repair Framework to Replicate Repair Approaches

In this section we define a unified repair framework (URF) that allows us to implement the behavior of the three repair approaches presented in Section 5.2.1.3.

Those repair approaches have different variabilities besides the repair operator each uses. The most important are:

1. fault localization used to detect suspicious source code component such as statements or methods;
2. the order these suspicious components are considered to apply a candidate fix;
3. the way a candidate fix is validated.

URF enables us to remove those free variables in the evaluation experiments. By removing free variables, our experiment focuses on the measurement of the strength of the pure

repair operators. For example, using different fault localization techniques is such a free variable. Qi et al. [106] have shown that the performance of an automatic repair approach varies according to the fault localization technique. Within URF, the use of the same fault localization technique removes the disparity between the different repair approaches under study.

Extending the framework, we implement GenProg, PAR and Debroy et al. approaches to repair Java programs. The main reasons for re-implementing them are: PAR and Debroy et al.'s approaches are not publicly available; and GenProg repairs C code and not Java. URF simplifies the implementation of these repair approaches. Most of the framework's components are shared between the three implementations. For example, the three approaches have the same implementation of fault localization and repair validation phases.

5.2.2.1 Unified Repair Framework Design

The unified framework is a generic search-based framework for repair and is composed of three phases.

5.2.2.1.1 Fault localization phase This phase defines a fault localization space formed by suspicious statements (statements suspected to contain a bug). Fault localization techniques compute a suspiciousness value for each statement in the program. Then, they use the calculated suspicious values to create a ranking of suspicious statements. The goal of this phase is twofold: to reduce first the search space size and second the time to navigate it.

The three approaches use fault localization techniques based on spectrum analysis. Spectrum based fault localization approaches execute test cases of a program and trace the software components (e.g., methods, lines) visited by those tests. The techniques use formulas to calculate the suspicious value of each component. These formulas usually take as input the collected traces and the test results. The suspicious value goes for 0 (low probability that the statement contains a bug) to 1 (high probability). GenProg and PAR use a formula presented by Weimer et al. in [12], while Debroy et al.'s work uses the Tarantula formula [82]. As the inputs (program code and the test suite) and output (suspicious statement list) of these formulas do not differ, these repair approaches could change the fault localization technique by another. Our framework uses the Ochiai formula [107] in the implementation of the three approaches.

The navigation of a fault localization space means to pick one suspicious statement from the space. This selected statement is modified according to the repair operators defined by an approach. Repair approaches apply different navigation strategies. For example, the space can be navigated: *a*) in order, for the most suspicious statement to the lowest; *b*) uniformly randomly: elements are uniform randomly selected; *c*) weighted randomly strategy: the probability to select a suspicious statement is proportional to its suspicious value. Our framework implements the weighted random strategy for the three approaches.

As we have mentioned, the same fault localization technique and navigation strategy for the implementations of the three repair approaches allows us to reduce the disparity of this phase. With our framework we unify these two free experimental variables across the three repair approach implementations.

5.2.2.1.2 Repair synthesis phase This phase receives one suspicious statement from the previous phase and returns a candidate repair, i.e., a patched program, to be evaluated. In this phase, the implementation of each approach defines the particular behavior to synthesize a candidate repair. For example, the synthesize phase of the mutation-based approach returns, given a suspicious *if condition* expression, a candidate repair that consists on the modification of one operator for the conditional expression. In Section 5.2.2.2 we discuss these particularities.

5.2.2.1.3 Repair validation The goal of this phase is to determine whether a candidate repair, generated in the previous phase, is valid or not. For that, the phase executes the test suite of a program. For sake of performance, the phase does two steps. First, it executes the originally failing test cases over the modified program. If these test cases now pass successfully, meaning that the bug is fixed, a regression test is executed to verify whether the candidate repair breaks the remaining functionalities. The regression test involves executing all test cases from the test suite. If none of them fails, the candidate repair is considered valid. If at least one test fails during the validation phase, the candidate repair is not valid and it is discarded.

The framework uses the same implementation of the phase for the three approaches. This phases define extension points to, for instance, integrate test case prioritization techniques [108] to reduce the time to execute regression validations. For instance, Ledru et al. [109, 110] present a test case prioritization technique based on string distances. The technique assigns high priority to a test case which is most different compared to those already prioritised. It relies on the information present in the test suite, so, it does not need test execution. Contrary, Qi et al. [81] present a prioritization technique applied in an automatic program repair approach. The technique extracts information from the repair process. It assists the patch validation process by improving the rate of invalid-patch detection in the context of automated program repair.

5.2.2.2 Specific Implementation Decisions

In this section we enumerate the decisions taken to implement GenProg, PAR and the mutation-based repair approach. They are essential to implement the approaches. Some decisions aim at clarifying those hidden or ambiguous issues in the original publications. We extend the repair synthesis phase of our framework (see Section 5.2.2.1) to encode the behavior of an approach. These implementations focus on repairing *if condition* defects. The section can be skipped if the implementation details are not relevant for the reader.

5.2.2.2.1 GenProg In GenProg the repair synthesis space is the product of two spaces: *operators kind space* and *fix ingredient space*. GenProg applies three kinds of operators: insert, remove and replace. Our implementation of GenProg applies one operator (replace) over *if condition* statements. This operator replaces a suspicious *if condition* (or a randomly selected sub-term from it) by another already written in the program.

We do not consider the remaining two operators (insert, remove) in our implementation. They are not relevant for the defect class under consideration. Both require defining additional assumptions beyond those originally proposed in GenProg. For instance, inserting one predicate in an existing *if condition* involves determining: the place to insert, and one

logical operator (AND, OR) as well. We exclude those repair operators to avoid including particular assumptions that alter the essence of the approach inside its implementation.

The fix ingredient space [12] contains all eligible statements for replace operator. For this defect class, these statements are conditional expressions. A *local* scope strategy includes statements from the class where the code is replaced, while *global* scope strategy includes statements from everywhere in the program. According to the literature, GenProg applies a global strategy: "A statement is chosen uniformly at random from anywhere in the program" [99, p.5]. However, as we show in Section 4.2.2, the local strategy allows repair approaches decreasing the search space, without neglecting "repair success potential". Consequently, we apply a local strategy. The fix ingredient space is formed by all predicates taken from *if condition* and *loops* statements located in the same class. Additionally, we add to the fix ingredient space all sub-terms included in conditional expression. For instance, given the existing $if((a > 0) \,||\, (b == null))$, the fix ingredient contains: the mentioned expression and two more terms: $(a > 0)$ and $(b == null)$.

To navigate the fix ingredient space, we apply a uniform random strategy: each element from this search space has the same probability to be selected.

As difference from GenProg, where its search algorithm is based on genetic programming, our implementation of GenProg is based on randomly search algorithms. That means, we randomly navigate the fault localization space to pick a suspicious statement, and the fix ingredient space to pick a candidate repair. Our decision is based in recent research [76] that shows the strength of GenProg is not due to the guidance of genetic programming, but relies on the strength of its operators.

5.2.2.2.2 PAR The repair synthesis space of PAR is also the product of two spaces: *operator kind space* and *fix ingredient space*. As we presented in Section 5.2.1, PAR defines two bug fix templates related to *if condition* statements that define the operators kind space: *a*) Expression replace for an *if condition*; and *b*) Expression added and removed: it inserts or removes a term of the *if condition*. To implement PAR we make the following assumptions.

We define the PAR's fix ingredient space as we do with GenProg. We have implemented the strategy to navigate this space in a different manner from PAR's paper. PAR's authors state the following strategy: "Same scope of the given fault location, sorted according to the distance". Instead, we extend PAR's strategy: We do a weighted random choice of a fix ingredient, where the weight is inversely proportional to the distance d (in number of lines) between the fix ingredient and the buggy statement.

The second PAR's fix template we consider in this work describes modifications in existing conditional statements. Let us first focus on the addition of terms and secondly in the removal.

To implement the "Expression Added" template we consider two assumptions. PAR's work does not specify both assumptions. Without them, it is not possible to implement those templates. The first one is the place where the new clause is inserted. The PAR paper does not discuss this point. The insertion can be placed: *a*) at the beginning of the *if condition* expression, e.g. given the predicate $if(b == null)$, adding the term c at the beginning results in $if((c)\ \&\&\ (b == null))$; or *b*) at the end, resulting $if((b == null)\ \&\&\ (c))$. The framework randomly chooses between both alternatives. The second assumption is the logical operator added to connect the new term with existing *if condition*. In the previous example we consider the insertion of the logical operator AND (&&). However, it could be used an-

other operator instead such as OR (||). PAR neither discusses this point. In particular, our framework randomly chooses one logical operator among *a*) *and*; *b*) *or*.

Regarding with the removal of a term, PAR randomly selects a term to remove. Our framework randomly removes one of the two terms related to a logical operator.

5.2.2.2.3 Debroy et al.'s mutation-based approach The framework defines a repair synthesis space (formed by mutants) by applying first-order mutation operators in one suspicious *if condition* statement. A first-order operator applies only one change in a statement. The synthesis space size depends on the number and the kinds of operators present in the statement. For example, given the suspicious if $if(a > b)$ the approach considers 6 candidates fix: one corresponding to the negation of the condition $!(a > b)$, the 5 remaining the replacement of the operator by another compatible operator such as $>=$ or $<$. The framework applies three kinds of mutation operators: Relational, Logical and Unary. They correspond to operators OP2, OP3 and OP8 defined in Debroy et al. (see Section 5.2.1.3). For Relational category the operators are six: $>, >=, <, =<, ==$, and $<>$; for Logical are two: OR, AND, and for Unary are two: *negation* and *positivation* (remove a negation operator). Finally, the framework randomly selects one mutant to be validated.

5.2.3 Summary

In this section we presented a framework to replicate three state-of-the art repair approaches. In the next following section present an experiment where we validate our evaluation methodology. We use the unified repair framework to repair defect from our *if condition* defect dataset presented in Section 5.1.5.

5.3 Empirical Evaluation Results of Repair Approaches Fixing *If Condition* Defects

In this section, we present an experiment to evaluate the ability of repairing *if condition* defects using the three repair approaches presented in Section 5.2.1. We implement those approaches with our repair framework presented in Section 5.2. This experiment allows us to concretely instantiate our evaluation methodology in order to validate it.

The motivation of this evaluation is twofold. First, we aim at identifying which approach is better i.e., repairs more defects for the defect class under consideration. We conduct the experiment by taking care of reducing the possible biases in the evaluation process. Second, we aim at obtaining the number of solutions that a repair search space is able to provide. To accomplish these goals, in Section 5.3.1 we present a set of measures. In Section 5.3.2 we present the research questions that guide our experiment. Then, in Section 5.3.3 we present the evaluation protocol, and finally in Section 5.3.4 we present the results of the evaluation.

5.3.1 Measures

In this section we present the measures we consider in the evaluation of repair approaches: *Repair diversity*, *Search Space Fertility*, *Failing test effort* and *Regression effort*. The two first measures are related to the measurement of repair capability of an approach, while the last two are related to the repair time measurement.

Repair diversity: We define it as the number of *different repairs* generated by one approach in the experiment. Two repairs are different if they are composed of different: *a*) variables and constants, or *b*) number of access to those variables, or *c*) kind and number of unary and binary operators.

Search Space Fertility: We define *fertility of a search space* as the proportion of their elements that are solution. In a high fertile space the majority of elements are solutions. Measuring fertility could be time demanding: one must navigate *all* the search space to know whether an element is a solution or not. In this work, we search for alternative measures to obtain an approximation for fertility. We propose one that measures the *ease* to find a solution in a repair space of one approach. In a high fertile space, solutions easily and rapidly blossom. On contrary, in a low fertile space, the navigation of the space is rough, and solutions are difficult to find. We measure the fertility of a repair space as the number of *distinct repairs found by one trial*.

Validation Effort: The validation of candidate repairs through the execution of test cases is the most time consuming phase in the automatic repair process [78]. For example, let us consider the version of Math project that contains defect $\#280$, included in our dataset. The 386 Java classes of the version are compiled in 9 seconds using Ant tool. Then, the failing test case is executed in 0.051 seconds.[26] Finally, we execute the regression test, i.e. all the cases from the test suite, in 2.55 minutes. The regression validation takes much more time than the compilation and failing case validation steps. As the execution of validation is an expensive phase, in this experiment we measure the effort to find a repair in terms of the number of times a validation process is executed before to find a repair.

We define two kinds of effort, one for each validation step defined in 5.2.2.1.3: *Failing test effort* and *Regression effort*.

Failing test effort measures the effort to find a candidate repair that passes all the failing test cases (those that initially fail due to the defect presence). This effort is measured in terms of number of times the *failing test cases* validation step is executed. Let us exemplify this measure: a repair approach tries to repair a defect by first applying a candidate repair p_1 at location l. The modified version of a program is validated (through the failing test cases). If the candidate repair is valid i.e., solves the defect, the effort is *one*: it executes one time the validation phase. Otherwise, the approach continues by applying a new candidate repair p_2 at l. If the modified version is valid the effort to find repair p_2 is *two* due it executes two times validation phase: one to validate repair p_1, the remaining to validate p_2.

Regression effort is similar to the previous measure. It measures the number of times the *regression* validation is executed before to find a repair. For instance, *Regression effort* of two means that regression phase was executed two times. In the first execution the phase fails (at least one test case fails), in the second execution, the regression is successful (all the test cases pass).

Both validation effort measures are similar to NCP, presented by Qi et al. [106]. It measures the number of candidate repairs generated before a valid repair is found in the repair process. NCP measure includes candidate repairs that do not pass: failing test cases validation phase, or the regression validation phase. That means, their measure does not distinguish candidate repairs that fail the *failing test cases validation phase* from those that fail the *regression phase*. However, as we previously have shown, the execution times of those phases

[26] Experiment done in PC, OS: Windows 7, CPU: Intel i7, RAM: 4gb

have different magnitudes. We consider that is a disadvantage to gather in a same measure different kinds of validation phase. As consequence, we propose a measure for each phase: *Failing test effort* and *Regression effort*.

5.3.2 Evaluation Goals

The experiment is guided by the following research questions:

RQ 1: How many defects from our dataset each repair approach is able to fix?
We aim at knowing whether each of the evaluated repair approaches is able to fix *if condition* defects, and which of them is better repairing this defect class. Previously, nobody has studied the difficulty to fix *if condition* defects. The response will let us know whether the evaluated approaches target *if condition* defects and, by consequence, whether it is necessary to improve or propose new paradigms to increase the repairability of this defect class.

RQ 2: How many different repairs are found by each approach?
Through this question, we aim at knowing whether an approach is able to synthesize more than one repair for a given defect. In this case, software developers are able to choose the fix that, from their criteria, is more suitable to integrate into the program under repair. Moreover, as the evaluated approaches use test suites to validate candidate fixes, a high number of solutions could mean the test suite does not have enough quality to distinguish a valid candidate repair from a invalid candidate.

RQ 3: How fertile is the search space of an approach?
In addition to the number of solutions an approach is able to discover from a repair space (previous research question), we aim at knowing the ease to find them. It could be the case that an approach can find one repair for a defect faster than another repair for the same defect.

RQ 4: Which approach requires less validation effort to find a repair?
Through this research question we aim at knowing which approach repairs faster. Repairing faster involves saving computational resources and to be able to deploy a repair faster into a defective program.

RQ 5: What do generated patches look like?
We aim at knowing what are the syntactic changes done to fix *if conditional* defects. The response could allow the research community to focus on defining new repair operators that are capable of synthesizing those changes.

5.3.3 Evaluation Protocol

We use the unified repair framework presented in Section 5.2.2.1 to evaluate three automatic repair techniques: GenProg, PAR and the mutation-based by Debroy et al. We encode the particularities of each approach using the extension points that our framework offers as we described in Section 5.2.2.2.

We call a *trial* the execution of a technique to repair a given defect. A trial executes n *iterations*, where each iteration consists in: *a*) selecting one suspicious statement (see 5.2.2.1.1); *b*) applying a candidate repair on it (see 5.2.2.2), *c*) compiling the repaired application; and, *d*) if it compiles, validating the candidate repair applying the two validation phases (see

5.2.2.1.3). Our experiment consists in the execution of t trials for each tuple repair technique-defect. We have maximum values on the number of trials and iterations. In particular, we execute 100 trials with 50 iterations each.

We evaluate the repair technique over defects from the dataset defined in Section 5.1.5. Our framework generated candidate repairs for 17 out of 19 defects. For the two remaining defects, #904 and #947, both from Math project, the framework could not retrieve the list of suspicious statement due to limitations of the fault localization tool it uses. For instance, the fault localization tool does not produce an output when a bug such as #904 produces an infinite loop.

5.3.4 Evaluation Results

Table 5.2 shows a summary of repairs found by defect and repair approach. Let us describe the table. Column *Defect* includes the identifiers of issues repaired by at least one approach implemented in our framework. The unrepairable issues are not present in this table. Column *Method* includes the name of the three repair approaches evaluated in this experiment. Column *Repair diversity* includes the total number of different patches generated by each approach in all the trial executed. A higher value is better. Column *Repair/trial* include the median and average number of repairs found per trial. Column *Validation effort* contains the median number of times the two validation phases (failing test and Regression sub-columns) are executed (see measures in Section 5.3.1) A lower value is better. Now, let us respond the research questions from the evaluation results.

5.3.4.1 RQ 1: How many defects from our dataset each repair approach is able to fix?

We summarize the number of *different repairs* generated by each approach in all the trial executed. The existing approaches could find repairs for 4 of 17 defects: #280, #288, #309, #340 from Math project, and #428 from Lang. PAR could repair the 5 mentioned defects, obtaining a repair efficacy of 29% (5/17) over the evaluated defects. GenProg could repair 4/17 (23%) and Debroy et al. repaired 3/17 (17%). The rest of the defects (12/17) of that dataset, that represent the same defect class, remain unrepaired. As we have seen before, in theory, the three approaches are able to repair defect from *if condition defect class*. However, in practice, the result of this experiment shows that the repairability of this defect class using these approaches is low: 5/17 (29%).

5.3.4.2 RQ 2: How many different repairs are found by each approach?

From column *Repair Diversity* of Table 5.2 we can answer this question: PAR generates 42 different repairs (32, 3, 3, 1 and 3 repairs for defects #280, #288, #309, #340 and #428, respectively); GenProg 15 (10, 0, 1, 1 and 3, repairs respectively); the Debroy et al. 4 (1, 2, 1, 0 and 0 repairs respectively).

PAR generates more different repairs for the 5 defects. Surprisingly, PAR and GenProg generate a considerable number of repairs for defect #280 (32 and 10, respectively). For example, a trial of GenProg found 5 repairs of this defect: three fix the defect replacing the *if condition* expression by another that uses different variables than the buggy version. We suspect the abundance of repairs could be related to the quality of the test suite. A low quality test suite could eventually consider a candidate repair as valid when it is not. This

Defect	Method	Distinct repairs	Repairs/trial		Validation effort (median)	
			Median	Avg	Failing test	Regression
#280	GenProg	10	5	4.9	4	1
#280	Debroy et al.	1	1	0.98	20	1
#280	PAR	32	2	1.87	11	1
#288	GenProg	0	-	-	-	-
#288	Debroy et al.	2	1	1.33	19	1
#288	PAR	3	0	0.23	7	1
#309	GenProg	1	1	0.91	3	1
#309	Debroy et al.	1	1	0.92	2	1
#309	PAR	3	0	0.45	3	1
#340	GenProg	1	0	0.07	6	1
#340	Debroy et al.	0	-	-	-	-
#340	PAR	1	1	0.98	3	1
#428	GenProg	3	1	0.82	1	1
#428	Debroy et al.	0	-	-	-	-
#428	PAR	3	0	0.16	1	1

Table 5.2: Summary of repairs generated for each approach. Higher "distinct repairs" and "Median and Avg Repairs/trial" are better. Lower validation effort is better. According to this dataset and evaluation protocol, there is no clearly better approach for repairing if conditional defects.

result triggers new challenges for our short-term research, such as researching about, for instance, alternative ways to measure the correctness of repairs or the quality of test cases.

5.3.4.3 RQ 3: How fertile is the search space of an approach?

As we mention in Section 5.3.1 we measure the space fertility as the *ease* to find a solution in the repair space of one approach. We measure the fertility of a repair space as the number of distinct repairs found by a trial. Columns *Median repairs/trial* and *Avg repairs/trial* show the median and average number of distinct repairs found by a trial, respectively. We could not determine a tendency from the results. Depending on the defect analyzed, the approach that finds more fixes per trial is different: for defect #288 is Debroy et al., for #428 is GenProg and for #340 is PAR. For example, let us consider defect #340, where PAR and GenProg both generate one repair. On the one hand, PAR finds the repair in almost all trials (Avg 0.98). On the other hand, GenProg *rarely* finds the repair in one trial (Avg 0.07). That means that, for that issue, the search space of GenProg is less fertile.

In our opinion, both *repair diversity* and *fertility* (i.e. Avg repairs/trial) measures are useful in the comparison of the approaches repairability. When diversity value is similar to fertility means the navigation of the space is straightforward and *repetitive*. Each trial produces

the same result (eventually they differ on the iteration that find a solution). When diversity is much greater than fertility means the navigation of the space is more *unpredictable*: the trials return different results. In this case, the final number of solutions found depends on the experiment configuration parameters such as the number of *trials* and *iterations*. As the result of trials is different, adding more trials in the experiment could eventually bring undiscovered solutions. As consequence, when researches compare two approaches, through experiments with those limitations, both metrics could be an indicator that the comparison is not fair.

5.3.4.4 RQ 4: Which approach requires less validation effort to find a repair?

Table 5.2 shows the validation effort done by each approach to repair defects. Column *Validation effort* presents two sub-columns *Failing test* and *Regression* that show the median number of times that the validation of failing test case and regression validations are executed before to find a repair, respectively.

We observe that, for defect #280 and #288, GenProg and PAR require less *failing test* validation effort to find a repair than Debroy et al., while the effort to repair defect #309 is almost the same (median 2 for Debroy et al. against 3 for the two remaining). Remember that less validation effort means less time to find a repair. However, it is not possible to determine the best between GenProg and PAR: for defects #309 and #428 the effort is the same, for #280 GenProg requires less, while for #288 PAR fixes it but GenProg does not.

Regarding with the *regression* validation effort, Table 5.2 shows that the median effort is always 1. We found that, for the 5 repaired defects, each time a candidate repair passes the failing test cases, then it passes the regression test. We observe that for each of the 61 patches found in the experiment (See Column *Repair Diversity* in Table 5.2). However, this situation does not happen for unrepairable defect. For defect #691 from our dataset, GenProg found a modification (a candidate repair) that produced the failing test cases pass. Then, the candidate repair could not pass the regression validation: there were test cases that fail with the modification. The candidate repair breaks functionality, as consequence, it is not considered solution for the defect (and that why it does not appear in Table 5.2).

Splitting the validation effort analysis in two measures allows us better understanding the validation of candidate repairs. Once a candidate patch passes the failing test phase, it would high probability passes also the regression and, by consequence, the candidate patch becomes a solution. This result implies that, by only executing one test case (the failing), one can estimate with high confidence whether a candidate patch is a solution for a bug or not.

5.3.4.5 RQ 5: What do generated patches look like?

In this section we analyze what patches generated by the evaluated repair approaches look like. We syntactically compare each *if condition* with the defect and it fixed version.

Table 5.3 presents changes that the generated patches introduce (rows), and the evaluated repair approaches (columns). Each cell (at column i, row j) contains the issues that were fixed by approach of column i through a patch that introduces changes described in row j. For example, issue #340 is fixed by both GenProg and PAR by removing a term in the buggy *if condition*.

From the first two rows of the table we observe two kinds of changes over binary operators: one change the relational operator $>=$ to $>$, another from $<$ to $=<$. The remaining rows group *if condition* repairs that: a) replace the entire *if condition* or b) add/remove terms

in an existing *if condition*. The code corresponding to the fix can introduce new variables, existing variables or combination of both. For instance, given a buggy expression $i.length()$ the fixed version i != $null$ && $i.length()$ introduces an existing variable in the expression (i). On contrary, given a buggy expression $i.length()$ the fixed version $j.isBoolean()$ && $i.length()$ introduces a new variable in the expression (j). Moreover, some rows are divided according to: a) the connector class *c* used to join an added term with the existing (and bug) expression (i.e. "AND", "OR") and b) the positions *p* where the added term is located ("before" or "after" the existing expression). For example, giving the buggy expression $i.length()$ the fix i != $null$ && $i.length()$ adds a new term (i != $null$) "before" the existing expression and it uses the "AND" (&&) logical operator.

The kind of change corresponds to the syntactic diff between the buggy *if condition* and the fixed version of this *if condition*. We recall it does not correspond to the *repair operation* that an approach applies to synthesize a candidate fix. For instance, for defect #280, GenProg and PAR synthesize the fix by replacing the whole buggy *if condition* $(fa * fb) >= 0.0$ by another condition in the scope $(fa * fb) > 0.0$. The syntactic difference between both is the relational operator in the *if condition* expression. As result, this repair is classified as "Replace relational operator < to <=", first row in Table 5.3 .

The result shows that there are issues such as #280 or #288 that can be fixed in different manners. For example, PAR generated two patches for issue #280 which add a term with new variables in the *if condition*: one adding at the beginning of the *if condition* joined by an AND relational operator, the other at the end of the *if condition*. For issue #288, the result shows that the issue is fixed with two different patches: one changes a relational operator from >= to > while the other changes from <to <=. The particularity of both patches is that affect two different locations in the faulty program i.e. two different *if condition*. This result shows us that it is possible to apply different valid patches over the same *if condition* or over different producing different syntactic programs. All of those programs contain the same behavior according to the program specification i.e., the test suite.

The analysis of the kind of bug has many applications. One is the definition of optimization strategies such as we presented in Section 4.1. A repair approach could learn from successfully synthesized repairs to improve the bug repairability. Another application could be the definition of studies that compare human fixes and automatically synthesized repairs. These studies could focus on, for instance, how developers reason during bug fixing activity, or how different are synthesized fixes from humans fixes.

5.3.5 Summary

The results of the empirical evaluation show that: *a*) the three approaches are capable to fix at least one defect from our dataset i.e., *if condition* defects; *b*) PAR fixes more defects from the dataset followed by GenProg; and *c*) GenProg and PAR require similar validation efforts to repair defects, Debroy et al. requires more validation effort than the two former. This means, GenProg and PAR repair faster the defect from our dataset.

In our opinion, this evaluation present several advantages compared with previous ones. First, the defect dataset used in the evaluation was built from a detailed methodology which aims at reducing biased defect dataset. Secondly, we introduce new measures to better understand the performance of automatic software repair approaches.

5.4 Conclusion

In this chapter we presented an evaluation of three state-of-the-art repair approaches. The challenge was to measure the repairability of repair approaches in a meaningful and unbiased manner. For that, we first defined a methodology to define evaluation datasets with a well-defined inclusion criterion. Using the methodology, we defined a dataset with 19 *if condition* defects from two open-source projects. Then, we defined a framework to replicate the selected repair approaches. Both the dataset and the framework are devised in order to minimize the risk of bias. Using this framework we could repair 5 out of 17 *if condition* defects from our dataset. From the results, we can conclude that the three evaluated approaches have a low repair efficacy repairing *if condition* defect class.

We conclude with some of the learned lessons in this chapter. We learned that it is possible to define a dataset for evaluations of repair approaches without taking in account particular features of the approaches under evaluation (that could affect the dataset creation). We learned the replication of repair approach involves clarifying assumptions about the hidden decisions each approach presents. We also know the features of search spaces differ between approaches: there are variations in their size and fertility. Finally, we learned that evaluation that compares repair approaches should take in account these features to not produce unbiased results.

Kind of change		GenProg	Par	Mutation-based
Replace relational operator >= to >		#280	#280	#280 #288
Replace relational operator < to <=		#309	-	#288 #309
Replacement expression with new variables		#280	#280	-
Replacement expression with new and existing variables		#280 #428	#288 #428	-
Replacement expression with existing variables		-	#288 #309	-
Add term with new variables to the expression	c: AND, p: After	-	#280	-
	c: AND, p: Before	-	#280	-
Add term with old variables to the expression	c: AND, p:Before	-	#280	-
	c: AND, p: After	-	#280	-
	c: OR, p: Before	-	#309	-
	c: OR, p: After	-	#309	-
Add term with new and old variables to the expression	c: OR, p: Before	-	#280 #288	-
	c: OR, p: After	#428	#280 #288 #428	-
Remove term in expression		#340	#340	-

Table 5.3: Kind of changes in if conditional involved by each patch. The rows are the syntactic change done to fix a bug, the columns are the evaluated repair approaches. A cell contains the issues resolved by each approach. Some of them are divided according to the connector *c* used to joint a term with a variable (i.e. "AND", "OR"), and the position *p* where the new term is located ("before" or "after" the existing expression.).

Chapter 6

Conclusion and Perspectives

This thesis concludes with a summary of the presented work and a discussion of future work.

6.1 Summary

Bug fixing is an activity for removing defects in software programs. An example of bug fix is the addition of an *if precondition* to check whether a variable is not null before accessing to it. Historically, bug fixing is done by software developers. Human fixing is a time consuming task for developers. To fix a bug, they have to reproduce the bug, study the symptoms, search for a candidate repair and finally validate it.

To decrease the time of bug fixing (and the related economic cost), several automatic software repair approaches have emerged to automatically synthesize bug fixes [12, 77, 5, 4, 76, 69, 78]. The proposed automatic repair approaches are evaluated by measuring the efficacy of repairing a set of defects. That means, given a defect dataset, the evaluation measures how many defects of the dataset an approach is able to repair. Unfortunately, bug fixing could be even difficult and expensive for automatic program repair approaches. The evaluations from the literature show that repair approaches are able to fix a portion of those defects. For instance, PAR fixes 27 out of 119 defects from its evaluation dataset [5]. Most of the state-of-the-art automatic software repair approaches do not use information from previous repairs done by developers to increase their defect repairability. For example, no approach considers the frequency of source code changes from the bug fixing activity.

The first major contribution of this thesis is a strategy to reduce the *time* to find a fix. It focuses on the most frequent kinds to repairs in order to find faster a solution. The strategy consumes information extracted from repairs done by developers. To obtain this information, we extract and analyze source code repairs done by developers from open-source projects. We present a technique to collect bug fix commits done by developers from the software history. The technique is based on the abstract syntax changes (AST) changes that commit introduces. Then, we present an approach to extract knowledge of *bug fix patterns* from the software history. A bug fix pattern is a set of source code changes that frequently appear together in the bug fixing activity. Our contributions are a method to formalize change patterns (such as bug fix patterns) and an another approach to collect instances of those change patterns from the software history. To validate those contributions, we first formalized bug

fix patterns from the literature [9], and then we measured their abundance in software version control systems of open-source projects.

The second contribution presented in this thesis is a strategy to reduce the repair time of one kind of automatic repair approach: *redundancy-based repair approach*. The strategy allows redundancy-based approaches to reduce their repair search space without losing repair strength.

Finally, we focus on the *evaluation of automatic repair approaches*. In the literature, no previous work has introduced defect datasets for *test suite-based repair approaches*, with a defined built criteria such as the defect classed included. Our contribution in this domain is a methodology to define defect datasets that minimize the possibility of biased results. This thesis contributes with a dataset of 19 *if condition* defects for further approaches evaluations. Then, to validate the methodology, we carried out an experiment that reproduces and compares the performance of three state-of-the-art repair approaches. We presented a framework to replicate automatic software repair approaches from the literature. The framework minimizes the variabilities (such as *fault localization* technique) between approaches under comparison, and it focuses on the strength of the repair operators defined by each approach. Through this experiment, we measured the repairability of *if condition defects*. The result of our repair evaluation shows that the evaluated approaches are able to repair a small fraction of *if condition* defects from our dataset.

6.2 Future Directions

In this section we present some ideas that we want to explore in the future.

6.2.1 Study of Software Evolution

In this thesis we focus on open-source projects such as those from Apache Software Foundation[27]. In future work we aim at replicating this experiment in commercial projects. One of the main goals of this future work is to know how bug fixing activity is done in these projects and what are the main differences compared to open-source projects.

In Section 3.4 we analyzed bug fix patterns from Pan et al.'s [9] bug fix pattern catalog. As the authors of this catalog state, a fraction of bug fix commits from open-source projects could not be classified. That means, no instance of their bug fix patterns was detected in those commits. In this thesis we presented additional bug fix patterns that complement this catalog. In future work we plan to automatically discover bug fix patterns from version control systems. We also aim at discovering two kinds of bug fix patterns: *horizontal* bug fix patterns are those patterns independent of the application; and *vertical* bug fix patterns are those patterns particular of one specific application.

6.2.1.1 Change Pattern Formalization

In Chapter 3 we defined a structure to formalize bug fix patterns. This structure allows us to define change patterns using the change taxonomy presented by Fluri et al. [22]. For example, the formalization allows us to specify a pattern that describes the addition of an

[27] http://www.apache.org/

if precondition and a *method invocation* inside the *if block*. For short term future work we aim extending the formalization to include information about the context that surrounds a change. We explain this limitation of our current pattern formalization in Section 3.4.8.2. For instance, a change pattern that specifies the addition of an *if condition* just before an existing *method invocation*. This extension would allow us to encode bug fix patterns from the literature such as the pattern "Addition of Method invocation in sequence of Method Calls" from Pan et al. [9] that cannot be encoded using the current formalization.

Our change pattern formalization uses the change taxonomy presented by Fluri et al. [22], which includes 41 changes over 147 source code entities such as "assignment" or "method invocation". We plan to use a new taxonomy to define finer grain changes over finer grain source code entities. Hence, a pattern formalization that uses a finer-grain change taxonomy would formalize more precise and descriptive change patterns. For example, we could define a change pattern that specifies changes in the left part of an assignment. Then, using our method presented in Chapter 3, one is able to measure the abundance of those finer grain change patterns.

6.2.2 Repair Approaches Design

In Chapter 4 we presented two strategies to optimize the search of repairs in a search space. We theoretically validated that our strategies improve the repairability of fixes. In the future we plan to add those strategies in repair approaches, in particular, into our unified repair framework presented in Chapter 5.

In Chapter 5 we present a framework that allows us to replicate repair approaches from the literature. In future work we plan to implement other *test suite-based* repair approaches such as Nopol [104]. Moreover, we plan to extend existing phases of our framework. For example, we aim at adding a technique of test case prioritization [108, 111] to reduce the cost of regression testing such as that one presented by Ledru et al. based on string distances [109, 110].

Our framework implements *evolutionary computing*. JAFF [69], GenProg [12] and PAR [5] repair approaches follow this paradigm. These approaches have a function called *fitness function*. In automatic repair context, it measure the distance between a candidate repair and a repair that is solution. The output of this function in GenProg and PAR is expressed in the number of failing test cases. The output zero means the evaluated program completely fulfills the specification. We plan to define a new fitness function, which will be capable to measure the distance to the solution in a finer grain than the number of test cases. We have the intuition that this kind of new function could help to find faster a solution in the repair search space.

The repair approaches that we analyzed in this thesis rely on test suite as correctness oracles. The test suites are used as a proxy of the specification. If all test cases pass, it means the evaluated program fulfills the program specification. We observe that quality of a test-suite impacts on the quality of the repair. In future work we plan to measure the quality of test suite from the automatic software repair perspective.

Current repair works relies on test suite as proxy of the program specification. A long-term research direction is the study of alternative mechanisms for validating candidates repair approaches.

6.2.3 Datasets and Repair Approaches Evaluations

In this thesis we presented a dataset of *if condition* defects for evaluating test suite-based repair approaches. In the short term future work we plan to define additional datasets for other defects such as *missing method invocations*, or *defects in loop conditions*. Then, we plan to analyze the repairability of the corresponding defect classes of the defined datasets using existing repair approaches. The result of those experiments would allow the research community to focus on those defect classes that are difficult to repair.

Appendix A

Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing

While the paper contains highlighted pieces of data, this appendix contains the whole data. The empirical results are computed from 62179 versioning transactions with at least one modified Java file of the repositories of Argouml, Columba, Jboss, Jhotdraw, Log4j, org.eclipse.ui.workbench, Struts, Carol, Dnsjava, Jedit, Junit, org.eclipse.jdt.core, Scarab and Tomcat [92].

A.1 Mathematical Formula for Computing the Median Number of Repair of MCShaper

Let's consider a set of N distinct repair actions $X_{i\in\{1,..,N\}}$. Each repair action has an occurrence probability p_i, ($\sum_i p_i = 1$) Let's define an "attempt" (drawing) consisting of n repair actions (one n-tuple, with the possibility that the some X_i are present more than once in the n-tuple, n is fixed. The question we pose is: *What is the median number of attempts for drawing a given n-tuple Y?* The response is obtained as follows :

First, we are interested in the following probability:

- probability of drawing Y on the 1er attempt is $P_1(Y)$
- probability of drawing Y on the 2d attempt is $P_2(Y)$
- ..
- probability of drawing Y on the K attempt is $P_K(Y)$

Let's assume that we know p, the probability of drawing p with a single draw. Then we have $P_2(Y) = (1-p)p$ which means that we don't draw Y on the first attempt but we draw it on the second. The formula, for recurrence, is $P_k(y) = p(1-p)^{k-1}$ After k attempts, the probability of drawing Y is:

$$S_K(Y) = P_1(Y) + P_2(Y) + .. + P_K(Y)$$

$$S_K(Y) = \sum_{k=1}^{k=K} P_k(Y)$$

$$S_K(Y) = \sum_{k=1}^{k=K} p(1-p)^{k-1}.$$

The median number of attempts to draw Y is k^* such that $S_{k^*}(Y) \geq 0,5$. So, we calculate S_k iteratively stopping when that condition is accomplished.

p is the probability of drawing an ordered tuple with repetition. It is $p = x \times \times \Pi_i p(i)$, where x is the number of equivalent drawings. For ordered tuple with repetition, x is obtained with the multinomial theorem [112, p.73]:

$$x = \binom{n}{e_1, e_2, ..., e_m} = \frac{n!}{\Pi_{j=1}^{j=m}(e_j!)}$$

(e_j is the number of occurrences of element j inside Y and m the number of uniques elements in Y).

Finally,

$$p = \binom{n}{e_1, e_2, ..., e_m} \times \Pi_{r \in R} P_{\mathcal{P}}(r)$$

Let's illustrate with a concrete example: $N = 5, p_{X1} = 0,1, p_{X2} = 0,1, p_{X3} = 0,2, p_{X4} = 0,2, p_{X5} = 0,4$. To find $Y = (X1, X3, X5)$: $p = 6 \times 0,1 \times 0,2 \times 0,4$, and $S_k \geq 0.5$ for k = 15 attempts. To find $Y = (X1, X3, X3)$: $p = 3 \times 0,1 \times 0,2 \times 0,2$, and $S_k \geq 0.5$ for k = 58 attempts.

We gratefully thank Ph. Preux for his help in getting this formula right.

A.2 Empirical results

Table A.2: The Semantic Changes of Change Model CTET Represented Among 62179 Versioning Transactions of Java Code.

Change Action	#changes.	Proba.
Statement_insert of Method_invocation	83046	6,9
Statement_insert of If_statement	79166	6,6
Statement_update of Method_invocation	76023	6,4
Statement_delete of Method_invocation	65357	5,5
Statement_delete of If_statement	59336	5
Statement_insert of Variable_declaration_statement	54951	4,6
Statement_insert of Assignment	49222	4,1
Additional_functionality of Method	49192	4,1
Statement_delete of Variable_declaration_statement	44519	3,7
Statement_update of Variable_declaration_statement	41838	3,5

Statement_delete of Assignment	41281	3,5
Condition_expression_change of If_statement	40415	3,4
Statement_update of Assignment	34802	2,9
Additional_object_state of Attribute	29328	2,5
Removed_functionality of Method	26172	2,2
Statement_insert of Return_statement	24184	2
Statement_parent_change of Method_invocation	21010	1,8
Statement_delete of Return_statement	20880	1,7
Alternative_part_insert of Else_statement	20227	1,7
Alternative_part_delete of Else_statement	17197	1,4
Removed_object_state of Attribute	16445	1,4
Statement_update of Return_statement	15132	1,3
Statement_ordering_change of Method_invocation	14267	1,2
Statement_parent_change of If_statement	12399	1
Statement_insert of Switch_case	10927	0,9
Statement_parent_change of Assignment	9851	0,8
Statement_parent_change of Variable_declaration_statement	9818	0,8
Statement_parent_change of Return_statement	9160	0,8
Statement_delete of Switch_case	8787	0,7
Statement_ordering_change of Variable_declaration_statement	8740	0,7
Statement_insert of Catch_clause	8708	0,7
Statement_ordering_change of Break_statement	8685	0,7
Parameter_insert of Single_variable_declaration	8609	0,7
Statement_ordering_change of Switch_case	8383	0,7
Statement_delete of Catch_clause	7927	0,7
Statement_insert of Try_statement	7489	0,6
Statement_insert of For_statement	7109	0,6
Statement_ordering_change of Assignment	7084	0,6
Decreasing_accessibility_change of Modifier	6772	0,6
Statement_delete of Try_statement	6618	0,6
Statement_delete of For_statement	6407	0,5
Parameter_type_change of Simple_type	6167	0,5
Parameter_delete of Single_variable_declaration	6048	0,5
Statement_ordering_change of If_statement	5637	0,5
Statement_insert of Throw_statement	5519	0,5
Method_renaming of Method_declaration	4931	0,4
Statement_insert of Break_statement	4767	0,4
Attribute_renaming of Field_declaration	4730	0,4
Increasing_accessibility_change of Modifier	4562	0,4
Parameter_renaming of Single_variable_declaration	4296	0,4
Statement_delete of Throw_statement	3876	0,3
Return_type_change of Simple_type	3413	0,3
Statement_insert of While_statement	3253	0,3
Statement_delete of Break_statement	3234	0,3
Attribute_type_change of Simple_type	3063	0,3
Statement_delete of While_statement	2817	0,2
Statement_update of Throw_statement	2807	0,2

Statement_update of Super_constructor_invocation	2379	0,2
Statement_ordering_change of Return_statement	2168	0,2
Statement_parent_change of Break_statement	1921	0,2
Statement_insert of Switch_statement	1832	0,2
Parent_class_change of Simple_type	1829	0,2
Statement_update of Switch_case	1634	0,1
Parameter_ordering_change of Single_variable_declaration	1606	0,1
Statement_insert of Continue_statement	1597	0,1
Parent_interface_insert of Simple_type	1515	0,1
Condition_expression_change of For_statement	1407	0,1
Statement_delete of Switch_statement	1394	0,1
Unclassified_change of Modifier	1334	0,1
Statement_parent_change of For_statement	1295	0,1
Statement_parent_change of Continue_statement	1221	0,1
Parent_interface_delete of Simple_type	1189	0,1
Additional_class of Class	1175	0,1
Statement_delete of Continue_statement	1081	0,1
Removing_attribute_modifiability of Modifier	1027	0,1
Statement_delete of Super_constructor_invocation	1011	0,1
Statement_insert of Super_constructor_invocation	941	0,1
Adding_attribute_modifiability of Modifier	916	0,1
Condition_expression_change of While_statement	786	0,1
Removed_class of Class	682	0,1
Statement_insert of Synchronized_statement	666	0,1
Statement_parent_change of Try_statement	642	0,1
Statement_update of Class_instance_creation	623	0,1
Return_type_change of Primitive_type	615	0,1
Statement_insert of Super_method_invocation	607	0,1
Removing_method_overridability of Modifier	603	0,1
Statement_parent_change of Throw_statement	577	0
Adding_method_overridability of Modifier	548	0
Statement_parent_change of Switch_case	536	0
Statement_parent_change of While_statement	526	0
Parameter_type_change of Primitive_type	513	0
Statement_insert of Constructor_invocation	465	0
Statement_ordering_change of Catch_clause	458	0
Statement_delete of Class_instance_creation	454	0
Statement_update of Switch_statement	450	0
Statement_insert of Labeled_statement	432	0
Statement_ordering_change of For_statement	430	0
Statement_update of Catch_clause	426	0
Parent_class_insert of Simple_type	411	0
Attribute_type_change of Primitive_type	411	0
Parent_interface_change of Simple_type	411	0
Statement_delete of Synchronized_statement	403	0
Statement_insert of Class_instance_creation	394	0
Statement_delete of Super_method_invocation	344	0

Statement_delete of Labeled_statement	326	0
Removing_class_derivability of Modifier	303	0
Statement_ordering_change of Continue_statement	284	0
Statement_update of Super_method_invocation	280	0
Return_type_delete of Simple_type	277	0
Statement_delete of Constructor_invocation	277	0
Return_type_insert of Simple_type	276	0
Statement_ordering_change of Try_statement	276	0
Statement_update of Constructor_invocation	258	0
Return_type_insert of Primitive_type	213	0
Parent_class_delete of Simple_type	177	0
Statement_parent_change of Switch_statement	175	0
Statement_insert of Do_statement	171	0
Return_type_delete of Primitive_type	167	0
Statement_parent_change of Super_method_invocation	164	0
Statement_ordering_change of Throw_statement	161	0
Statement_update of Synchronized_statement	159	0
Statement_delete of Assert_statement	152	0
Statement_delete of Do_statement	146	0
Statement_ordering_change of While_statement	143	0
Statement_update of Break_statement	124	0
Statement_update of Labeled_statement	120	0
Unclassified_change of If_statement	119	0
Adding_class_derivability of Modifier	99	0
Statement_ordering_change of Super_method_invocation	95	0
Condition_expression_change of Do_statement	86	0
Unclassified_change of Variable_declaration_statement	86	0
Unclassified_change of Return_statement	84	0
Unclassified_change of Assignment	75	0
Statement_ordering_change of Switch_statement	67	0
Statement_insert of Assert_statement	59	0
Unclassified_change of Method_invocation	56	0
Statement_parent_change of Labeled_statement	48	0
Statement_insert of Enhanced_for_statement	43	0
Statement_parent_change of Synchronized_statement	41	0
Class_renaming of Type_declaration	35	0
Statement_ordering_change of Class_instance_creation	32	0
Unclassified_change of Else_statement	28	0
Statement_ordering_change of Synchronized_statement	27	0
Unclassified_change of Line_comment	23	0
Statement_parent_change of Do_statement	21	0
Statement_parent_change of Class_instance_creation	17	0
Statement_delete of Enhanced_for_statement	16	0
Unclassified_change of For_statement	16	0
Statement_ordering_change of Labeled_statement	12	0
Unclassified_change of Switch_case	11	0
Unclassified_change of Catch_clause	10	0

Unclassified_change of While_statement	10	0
Unclassified_change of Try_statement	10	0
Unclassified_change of Block_comment	9	0
Condition_expression_change of Enhanced_for_statement	8	0
Unclassified_change of Switch_statement	6	0
Statement_update of Continue_statement	6	0
Unclassified_change of Empty_statement	4	0
Unclassified_change of Postfix_expression	4	0
Statement_parent_change of Assert_statement	3	0
Parameter_type_change of Parameterized_type	3	0
Unclassified_change of Super_constructor_invocation	2	0
Statement_ordering_change of Enhanced_for_statement	2	0
Parent_class_delete of Parameterized_type	2	0
Statement_ordering_change of Do_statement	2	0
Unclassified_change of Break_statement	2	0
Statement_update of Assert_statement	2	0
Return_type_insert of Parameterized_type	2	0
Unclassified_change of Type_literal	1	0
Return_type_change of Parameterized_type	1	0
Parent_class_insert of Parameterized_type	1	0
Unclassified_change of Constructor_invocation	1	0
Unclassified_change of Throw_statement	1	0
Total	1196385	

Table A.6: CTET Repair Actions Types and Probability χ_i for Different Heuristics to Build Versioning Transaction Bags.

Item	ALL	1-LC	1-SC	BFP	20-SC
Statement_insert-Method_invocation	0.0694	0.0983	0.0897	0.0800	0.0881
Statement_insert-If_statement	0.0662	0.0503	0.0000	0.0776	0.0806
Statement_update-Method_invocation	0.0635	0.1311	0.1502	0.0549	0.0673
Statement_delete-Method_invocation	0.0546	0.0557	0.0439	0.0586	0.0454
Statement_delete-If_statement	0.0496	0.0232	0.0000	0.0505	0.0347
Statement_insert-Variable_declaration_statement	0.0459	0.0083	0.0009	0.0509	0.0536
Statement_insert-Assignment	0.0411	0.0338	0.0210	0.0456	0.0445
Additional_functionality-Method	0.0411	0.0013	0.1381	0.0407	0.0669
Statement_delete-Variable_declaration_statement	0.0372	0.0137	0.0104	0.0368	0.0283
Statement_update-Variable_declaration_statement	0.0350	0.0908	0.0947	0.0283	0.0431
Statement_delete-Assignment	0.0345	0.0132	0.0066	0.0303	0.0206
Condition_expression_change-If_statement	0.0338	0.1223	0.1251	0.0277	0.0445
Statement_update-Assignment	0.0291	0.0599	0.0671	0.0232	0.0282
Additional_object_state-Attribute	0.0245	0.0072	0.0253	0.0245	0.0299
Removed_functionality-Method	0.0219	0.0021	0.0451	0.0167	0.0222
Statement_insert-Return_statement	0.0202	0.0325	0.0017	0.0189	0.0252
Statement_parent_change-Method_invocation	0.0176	0.0204	0.0031	0.0219	0.0233
Statement_delete-Return_statement	0.0175	0.0183	0.0000	0.0151	0.0139

Alternative_part_insert-Else_statement	0.0169	0.0052	0.0000	0.0194	0.0171
Alternative_part_delete-Else_statement	0.0144	0.0054	0.0000	0.0150	0.0095
Removed_object_state-Attribute	0.0137	0.0077	0.0165	0.0114	0.0116
Statement_update-Return_statement	0.0126	0.0353	0.0501	0.0084	0.0135
Statement_ordering_change-Method_invocation	0.0119	0.0052	0.0092	0.0124	0.0099
Statement_parent_change-If_statement	0.0104	0.0083	0.0021	0.0116	0.0142
Statement_insert-Switch_case	0.0091	0.0031	0.0028	0.0104	0.0025
Statement_parent_change-Variable_declaration_statement	0.0082	0.0015	0.0014	0.0089	0.0102
Statement_parent_change-Assignment	0.0082	0.0121	0.0017	0.0100	0.0093
Statement_parent_change-Return_statement	0.0077	0.0157	0.0005	0.0077	0.0092
Statement_ordering_change-Variable_declaration_statement	0.0073	0.0015	0.0007	0.0079	0.0052
Statement_insert-Catch_clause	0.0073	0.0036	0.0002	0.0088	0.0071
Statement_ordering_change-Break_statement	0.0073	0.0015	0.0000	0.0131	0.0013
Statement_delete-Switch_case	0.0073	0.0005	0.0002	0.0076	0.0007
Parameter_insert-Single_variable_declaration	0.0072	0.0018	0.0007	0.0065	0.0053
Statement_ordering_change-Switch_case	0.0070	0.0000	0.0007	0.0110	0.0005
Statement_delete-Catch_clause	0.0066	0.0036	0.0000	0.0104	0.0043
Statement_insert-Try_statement	0.0063	0.0000	0.0000	0.0064	0.0056
Statement_insert-For_statement	0.0059	0.0010	0.0000	0.0061	0.0049
Statement_ordering_change-Assignment	0.0059	0.0031	0.0014	0.0060	0.0043
Decreasing_accessibility_change-Modifier	0.0057	0.0057	0.0071	0.0039	0.0048
Statement_delete-Try_statement	0.0055	0.0000	0.0000	0.0079	0.0033
Statement_delete-For_statement	0.0054	0.0010	0.0000	0.0053	0.0033
Parameter_type_change-Simple_type	0.0052	0.0031	0.0038	0.0026	0.0028
Parameter_delete-Single_variable_declaration	0.0051	0.0008	0.0000	0.0037	0.0026
Statement_ordering_change-If_statement	0.0047	0.0028	0.0043	0.0051	0.0043
Statement_insert-Throw_statement	0.0046	0.0008	0.0002	0.0037	0.0050
Method_renaming-Method_declaration	0.0041	0.0021	0.0021	0.0027	0.0030
Attribute_renaming-Field_declaration	0.0040	0.0008	0.0012	0.0026	0.0029
Statement_insert-Break_statement	0.0040	0.0023	0.0012	0.0040	0.0019
Increasing_accessibility_change-Modifier	0.0038	0.0124	0.0137	0.0031	0.0059
Parameter_renaming-Single_variable_declaration	0.0036	0.0010	0.0009	0.0020	0.0023
Statement_delete-Throw_statement	0.0032	0.0018	0.0005	0.0038	0.0023
Return_type_change-Simple_type	0.0029	0.0013	0.0014	0.0013	0.0018
Statement_delete-Break_statement	0.0027	0.0008	0.0000	0.0019	0.0006
Statement_insert-While_statement	0.0027	0.0010	0.0000	0.0029	0.0026
Attribute_type_change-Simple_type	0.0026	0.0021	0.0019	0.0013	0.0021
Statement_delete-While_statement	0.0024	0.0005	0.0000	0.0024	0.0017
Statement_update-Throw_statement	0.0023	0.0054	0.0061	0.0029	0.0027
Statement_update-Super_constructor_invocation	0.0020	0.0072	0.0099	0.0013	0.0022
Statement_ordering_change-Return_statement	0.0018	0.0034	0.0000	0.0020	0.0014
Statement_parent_change-Break_statement	0.0016	0.0003	0.0002	0.0018	0.0005
Statement_insert-Switch_statement	0.0015	0.0000	0.0000	0.0013	0.0004
Parent_class_change-Simple_type	0.0015	0.0026	0.0024	0.0014	0.0016
Statement_update-Switch_case	0.0014	0.0003	0.0002	0.0009	0.0007
Parent_interface_insert-Simple_type	0.0013	0.0041	0.0045	0.0009	0.0017
Parameter_ordering_change-Single_variable_declaration	0.0013	0.0000	0.0000	0.0009	0.0005

Statement_insert-Continue_statement	0.0013	0.0036	0.0002	0.0013	0.0015
Statement_delete-Switch_statement	0.0012	0.0000	0.0000	0.0008	0.0001
Condition_expression_change-For_statement	0.0012	0.0013	0.0014	0.0010	0.0013
Unclassified_change-Modifier	0.0011	0.0013	0.0019	0.0008	0.0010
Statement_parent_change-For_statement	0.0011	0.0008	0.0000	0.0012	0.0017
Parent_interface_delete-Simple_type	0.0010	0.0046	0.0045	0.0006	0.0012
Additional_class-Class	0.0010	0.0000	0.0014	0.0010	0.0018
Statement_parent_change-Continue_statement	0.0010	0.0028	0.0000	0.0010	0.0007
Statement_delete-Continue_statement	0.0009	0.0008	0.0000	0.0006	0.0005
Removing_attribute_modifiability-Modifier	0.0009	0.0013	0.0012	0.0007	0.0013
Statement_delete-Super_constructor_invocation	0.0008	0.0023	0.0002	0.0007	0.0010
Adding_attribute_modifiability-Modifier	0.0008	0.0003	0.0005	0.0002	0.0003
Statement_insert-Super_constructor_invocation	0.0008	0.0023	0.0002	0.0006	0.0009
Condition_expression_change-While_statement	0.0007	0.0018	0.0017	0.0004	0.0007
Statement_insert-Synchronized_statement	0.0006	0.0000	0.0000	0.0009	0.0008
Removed_class-Class	0.0006	0.0000	0.0026	0.0005	0.0006
Statement_insert-Super_method_invocation	0.0005	0.0023	0.0019	0.0005	0.0008
Adding_method_overridability-Modifier	0.0005	0.0003	0.0002	0.0012	0.0002
Removing_method_overridability-Modifier	0.0005	0.0003	0.0002	0.0004	0.0003
Statement_parent_change-Try_statement	0.0005	0.0000	0.0000	0.0006	0.0010
Statement_parent_change-Throw_statement	0.0005	0.0003	0.0000	0.0005	0.0007
Statement_update-Class_instance_creation	0.0005	0.0010	0.0014	0.0003	0.0002
Return_type_change-Primitive_type	0.0005	0.0000	0.0000	0.0001	0.0002
Statement_insert-Constructor_invocation	0.0004	0.0003	0.0000	0.0003	0.0006
Statement_delete-Class_instance_creation	0.0004	0.0005	0.0007	0.0002	0.0003
Statement_insert-Labeled_statement	0.0004	0.0000	0.0000	0.0003	0.0001
Statement_parent_change-While_statement	0.0004	0.0003	0.0000	0.0005	0.0006
Statement_ordering_change-Catch_clause	0.0004	0.0000	0.0000	0.0004	0.0005
Statement_parent_change-Switch_case	0.0004	0.0000	0.0000	0.0002	0.0000
Statement_ordering_change-For_statement	0.0004	0.0000	0.0000	0.0004	0.0004
Parameter_type_change-Primitive_type	0.0004	0.0000	0.0002	0.0001	0.0002
Statement_update-Switch_statement	0.0004	0.0000	0.0000	0.0001	0.0002
Statement_update-Catch_clause	0.0004	0.0013	0.0017	0.0003	0.0007
Parent_interface_change-Simple_type	0.0003	0.0005	0.0005	0.0002	0.0006
Statement_delete-Labeled_statement	0.0003	0.0010	0.0000	0.0002	0.0002
Statement_delete-Synchronized_statement	0.0003	0.0000	0.0000	0.0004	0.0004
Parent_class_insert-Simple_type	0.0003	0.0003	0.0002	0.0002	0.0002
Statement_insert-Class_instance_creation	0.0003	0.0010	0.0002	0.0002	0.0003
Statement_delete-Super_method_invocation	0.0003	0.0023	0.0009	0.0002	0.0004
Removing_class_derivability-Modifier	0.0003	0.0003	0.0005	0.0002	0.0002
Attribute_type_change-Primitive_type	0.0003	0.0000	0.0000	0.0001	0.0002
Return_type_insert-Simple_type	0.0002	0.0003	0.0002	0.0002	0.0002
Statement_delete-Constructor_invocation	0.0002	0.0005	0.0000	0.0001	0.0002
Return_type_delete-Simple_type	0.0002	0.0003	0.0000	0.0002	0.0001
Return_type_insert-Primitive_type	0.0002	0.0000	0.0000	0.0001	0.0002
Statement_ordering_change-Continue_statement	0.0002	0.0000	0.0000	0.0003	0.0001
Statement_update-Super_method_invocation	0.0002	0.0005	0.0005	0.0002	0.0003

Statement_ordering_change-Try_statement	0.0002	0.0000	0.0000	0.0002	0.0003
Statement_update-Constructor_invocation	0.0002	0.0008	0.0007	0.0002	0.0003
Condition_expression_change-Do_statement	0.0001	0.0000	0.0000	0.0000	0.0001
Statement_ordering_change-Switch_statement	0.0001	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Variable_declaration_statement	0.0001	0.0000	0.0000	0.0002	0.0000
Parent_class_delete-Simple_type	0.0001	0.0000	0.0005	0.0001	0.0002
Return_type_delete-Primitive_type	0.0001	0.0000	0.0000	0.0001	0.0001
Statement_ordering_change-Throw_statement	0.0001	0.0000	0.0000	0.0001	0.0002
Statement_update-Labeled_statement	0.0001	0.0000	0.0000	0.0001	0.0000
Unclassified_change-If_statement	0.0001	0.0000	0.0000	0.0003	0.0000
Statement_ordering_change-Super_method_invocation	0.0001	0.0003	0.0007	0.0001	0.0002
Unclassified_change-Assignment	0.0001	0.0000	0.0000	0.0001	0.0000
Statement_update-Break_statement	0.0001	0.0000	0.0000	0.0002	0.0000
Statement_ordering_change-While_statement	0.0001	0.0000	0.0000	0.0002	0.0001
Unclassified_change-Return_statement	0.0001	0.0000	0.0000	0.0002	0.0000
Statement_delete-Do_statement	0.0001	0.0000	0.0000	0.0001	0.0001
Adding_class_derivability-Modifier	0.0001	0.0000	0.0000	0.0001	0.0001
Statement_parent_change-Switch_statement	0.0001	0.0000	0.0000	0.0002	0.0001
Statement_parent_change-Super_method_invocation	0.0001	0.0000	0.0000	0.0002	0.0002
Statement_insert-Do_statement	0.0001	0.0000	0.0000	0.0002	0.0002
Statement_update-Synchronized_statement	0.0001	0.0000	0.0000	0.0001	0.0001
Statement_delete-Assert_statement	0.0001	0.0000	0.0000	0.0000	0.0001
Statement_insert-Enhanced_for_statement	0.0000	0.0000	0.0000	0.0000	0.0001
Unclassified_change-Empty_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_update-Continue_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-For_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_ordering_change-Class_instance_creation	0.0000	0.0000	0.0000	0.0000	0.0000
Return_type_change-Parameterized_type	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Throw_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Constructor_invocation	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Type_literal	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Try_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_delete-Enhanced_for_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_parent_change-Do_statement	0.0000	0.0000	0.0000	0.0000	0.0001
Unclassified_change-Switch_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Parent_class_insert-Parameterized_type	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_update-Assert_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_ordering_change-Labeled_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Else_statement	0.0000	0.0000	0.0000	0.0001	0.0000
Statement_parent_change-Class_instance_creation	0.0000	0.0003	0.0000	0.0000	0.0000
Class_renaming-Type_declaration	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Super_constructor_invocation	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_ordering_change-Enhanced_for_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Catch_clause	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-While_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Break_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_parent_change-Synchronized_statement	0.0000	0.0000	0.0000	0.0000	0.0001

Statement_parent_change-Assert_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_insert-Assert_statement	0.0000	0.0000	0.0000	0.0001	0.0001
Unclassified_change-Switch_case	0.0000	0.0000	0.0000	0.0000	0.0000
Condition_expression_change-Enhanced_for_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Line_comment	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_ordering_change-Synchronized_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Parameter_type_change-Parameterized_type	0.0000	0.0000	0.0000	0.0000	0.0000
Parent_class_delete-Parameterized_type	0.0000	0.0000	0.0000	0.0000	0.0000
Statement_parent_change-Labeled_statement	0.0000	0.0000	0.0000	0.0000	0.0001
Statement_ordering_change-Do_statement	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Method_invocation	0.0000	0.0000	0.0000	0.0001	0.0000
Unclassified_change-Postfix_expression	0.0000	0.0000	0.0000	0.0000	0.0000
Unclassified_change-Block_comment	0.0000	0.0000	0.0000	0.0000	0.0000
Return_type_insert-Parameterized_type	0.0000	0.0000	0.0000	0.0000	0.0000

A.3 Bug Fix Survey Summary

The survey data is available at `https://sites.google.com/site/matiassebastianmartinez/dataset_survey_Martinez_et_al_2013.zip`.

Table A.1: The Frequency of Semantic Changes of Change Model CT Represented Among 62179 Versioning Transactions of Java Code.

Change Action	#changes	Proba.
Statement_insert	345548	28,9
Statement_delete	276643	23,1
Statement_update	177063	14,8
Statement_parent_change	69425	5,8
Statement_ordering_change	56953	4,8
Additional_functionality	49192	4,1
Condition_expression_change	42702	3,6
Additional_object_state	29328	2,5
Removed_functionality	26172	2,2
Alternative_part_insert	20227	1,7
Alternative_part_delete	17197	1,4
Removed_object_state	16445	1,4
Parameter_insert	8609	0,7
Decreasing_accessibility_change	6772	0,6
Parameter_type_change	6683	0,6
Parameter_delete	6048	0,5
Method_renaming	4931	0,4
Attribute_renaming	4730	0,4
Increasing_accessibility_change	4562	0,4
Parameter_renaming	4296	0,4
Return_type_change	4029	0,3
Attribute_type_change	3474	0,3
Unclassified_change	1892	0,2
Parent_class_change	1829	0,2
Parameter_ordering_change	1606	0,1
Parent_interface_insert	1515	0,1
Parent_interface_delete	1189	0,1
Additional_class	1175	0,1
Removing_attribute_modifiability	1027	0,1
Adding_attribute_modifiability	916	0,1
Removed_class	682	0,1
Removing_method_overridability	603	0,1
Adding_method_overridability	548	0
Return_type_insert	491	0
Return_type_delete	444	0
Parent_class_insert	412	0
Parent_interface_change	411	0
Removing_class_derivability	303	0
Parent_class_delete	179	0
Adding_class_derivability	99	0
Class_renaming	35	0
Total	1196385	

Table A.3: Spearman Correlation between the CT Change Action Probabilities of 14 Java Software Repositories. The majority is higher than 0.9, showing that the probability distribution over change actions is project-independent.

	dnsjava.cvs	columba	argouml	jboss	org.eclipse.jdt.core	org.eclipse.ui.workbench	tomcat.cvs	jEdit	struts.cvs	scarab	log4j.cvs	jhotdraw6	junit	carol
dnsjava.cvs	1.00	0.87	0.89	0.88	0.89	0.90	0.85	0.90	0.82	0.91	0.90	0.85	0.87	0.83
columba	0.87	1.00	0.94	0.91	0.85	0.91	0.84	0.88	0.91	0.92	0.90	0.91	0.85	0.88
argouml	0.89	0.94	1.00	0.95	0.89	0.94	0.85	0.90	0.87	0.93	0.94	0.90	0.85	0.92
jboss	0.88	0.91	0.95	1.00	0.91	0.96	0.84	0.90	0.87	0.92	0.97	0.87	0.86	0.94
org.eclipse.jdt.core	0.89	0.85	0.89	0.91	1.00	0.96	0.92	0.92	0.87	0.93	0.92	0.84	0.81	0.88
org.eclipse.ui.work	0.90	0.91	0.94	0.96	0.96	1.00	0.89	0.93	0.89	0.95	0.98	0.86	0.87	0.93
tomcat.cvs	0.85	0.84	0.85	0.84	0.92	0.89	1.00	0.90	0.83	0.89	0.87	0.87	0.83	0.80
jEdit	0.90	0.88	0.90	0.90	0.92	0.93	0.90	1.00	0.85	0.91	0.93	0.85	0.89	0.86
struts.cvs	0.82	0.91	0.87	0.87	0.87	0.89	0.83	0.85	1.00	0.90	0.87	0.83	0.81	0.88
scarab	0.91	0.92	0.93	0.92	0.93	0.95	0.89	0.91	0.90	1.00	0.94	0.88	0.87	0.90
log4j.cvs	0.90	0.90	0.94	0.97	0.92	0.98	0.87	0.93	0.87	0.94	1.00	0.86	0.91	0.93
jhotdraw6	0.85	0.91	0.90	0.87	0.84	0.86	0.87	0.85	0.83	0.88	0.86	1.00	0.85	0.84
junit	0.87	0.85	0.85	0.86	0.81	0.87	0.83	0.89	0.81	0.87	0.91	0.85	1.00	0.83
carol	0.83	0.88	0.92	0.94	0.88	0.93	0.80	0.86	0.88	0.90	0.93	0.84	0.83	1.00

Figure A.1: The distribution of the ranking difference for the most correlated project pair (workbench&log4j, Spearman correlation of 0.98) and the least correlated project pair (Tomcat&Carol, Spearman correlation of 0.80) in change model CT. The project pair workbench&log4j has more change actions with distance lower than 5 (9 vs. 7), and project pairs Tomcat&Carol has more changes actions with rank distance greater than 15 (6 vs. 0). This explains the difference in the Spearman correlation values. Overall, the shape of the distribution is very similar.

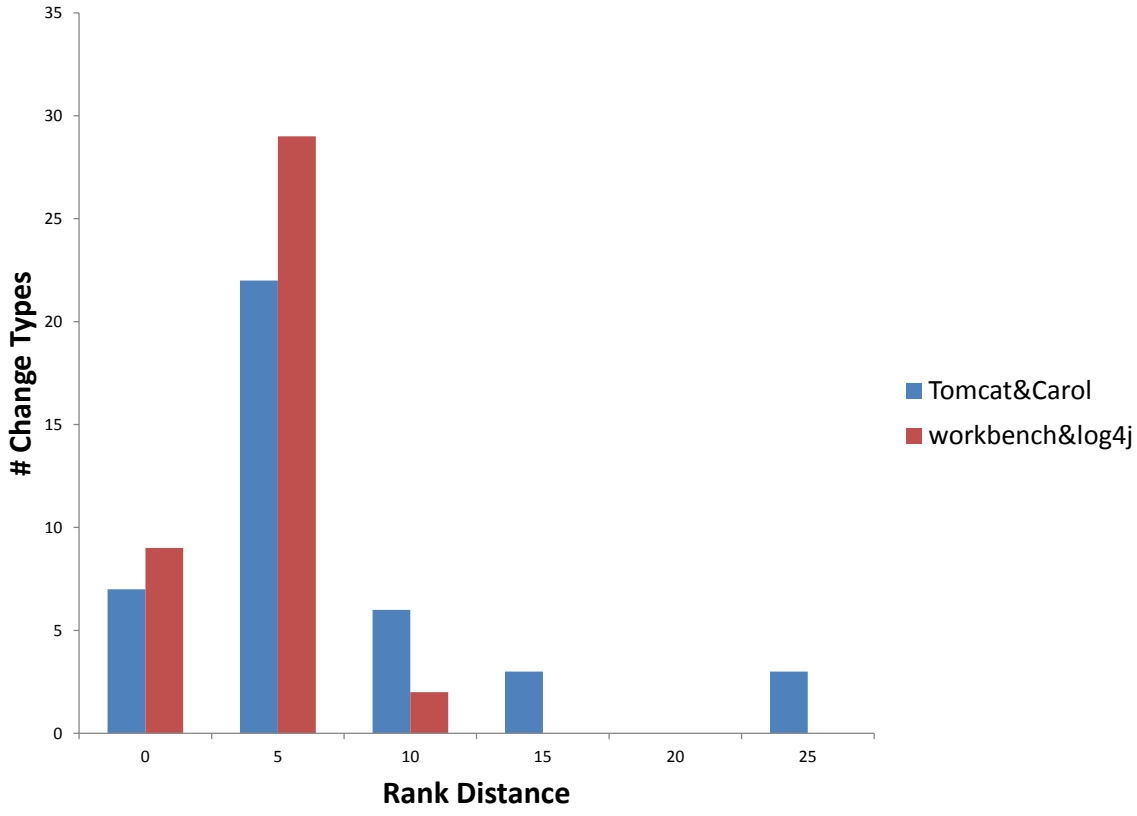

Table A.4: Spearman Correlation between the CTET Change Action Probabilities of 14 Java Software Repositories. They are slightly lower than those of change model CT but the probability distribution over change actions can still be considered as project-independent.

	dnsjava	columba	argouml	jboss	eclipse.jdt.core	eclipse.ui.workbench	tomcat	jEdit	struts	scarab	log4j	jhotdraw6	junit
dnsjava	1.00	0.84	0.83	0.89	0.86	0.87	0.88	0.87	0.84	0.85	0.87	0.79	0.73
columba	0.84	1.00	0.92	0.90	0.80	0.90	0.88	0.89	0.85	0.87	0.91	0.87	0.81
argouml	0.83	0.92	1.00	0.91	0.80	0.87	0.86	0.89	0.85	0.86	0.90	0.86	0.77
jboss	0.89	0.90	0.91	1.00	0.84	0.91	0.95	0.88	0.89	0.91	0.95	0.83	0.80
eclipse.jdt.core	0.86	0.80	0.80	0.84	1.00	0.83	0.87	0.89	0.81	0.78	0.86	0.73	0.64
eclipse.ui.workbench	0.87	0.90	0.87	0.91	0.83	1.00	0.91	0.89	0.82	0.89	0.91	0.84	0.76
tomcat	0.88	0.88	0.86	0.95	0.87	0.91	1.00	0.90	0.87	0.88	0.92	0.82	0.77
jEdit	0.87	0.89	0.89	0.88	0.89	0.89	0.90	1.00	0.85	0.85	0.89	0.80	0.73
struts	0.84	0.85	0.85	0.89	0.81	0.82	0.87	0.85	1.00	0.85	0.85	0.81	0.74
scarab	0.85	0.87	0.86	0.91	0.78	0.89	0.88	0.85	0.85	1.00	0.89	0.82	0.76
log4j	0.87	0.91	0.90	0.95	0.86	0.91	0.92	0.89	0.85	0.89	1.00	0.84	0.81
jhotdraw6	0.79	0.87	0.86	0.83	0.73	0.84	0.82	0.80	0.81	0.82	0.84	1.00	0.77
junit	0.73	0.81	0.77	0.80	0.64	0.76	0.77	0.73	0.74	0.76	0.81	0.77	1.00
carol	0.83	0.84	0.84	0.91	0.73	0.83	0.86	0.79	0.86	0.88	0.86	0.80	0.78

Table A.5: CT Repair Actions and Probability χ_i for Different Heuristics to Build Versioning Transaction Bags.

Item	ALL	1-LC	1-SC	BFP	20-SC	20-LC
Statement_insert	0.2888	0.2446	0.1204	0.3211	0.3273	0.3417
Statement_delete	0.2312	0.1398	0.0635	0.2340	0.1624	0.1776
Statement_update	0.1480	0.3336	0.3825	0.1213	0.1596	0.1514
Statement_parent_change	0.0580	0.0627	0.0090	0.0664	0.0719	0.0986
Statement_ordering_change	0.0476	0.0178	0.0170	0.0592	0.0286	0.0458
Additional_functionality	0.0411	0.0013	0.1381	0.0407	0.0669	0.0210
Condition_expression_change	0.0357	0.1254	0.1282	0.0291	0.0466	0.0551
Additional_object_state	0.0245	0.0072	0.0253	0.0245	0.0299	0.0156
Removed_functionality	0.0219	0.0021	0.0451	0.0167	0.0222	0.0070
Alternative_part_insert	0.0169	0.0052	0.0000	0.0194	0.0171	0.0220
Alternative_part_delete	0.0144	0.0054	0.0000	0.0150	0.0095	0.0131
Removed_object_state	0.0137	0.0077	0.0165	0.0114	0.0116	0.0071
Parameter_insert	0.0072	0.0018	0.0007	0.0065	0.0053	0.0050
Decreasing_accessibility_change	0.0057	0.0057	0.0071	0.0039	0.0048	0.0038
Parameter_type_change	0.0056	0.0031	0.0040	0.0026	0.0030	0.0040
Parameter_delete	0.0051	0.0008	0.0000	0.0037	0.0026	0.0022
Method_renaming	0.0041	0.0021	0.0021	0.0027	0.0030	0.0023
Attribute_renaming	0.0040	0.0008	0.0012	0.0026	0.0029	0.0022
Increasing_accessibility_change	0.0038	0.0124	0.0137	0.0031	0.0059	0.0086
Parameter_renaming	0.0036	0.0010	0.0009	0.0020	0.0023	0.0020
Return_type_change	0.0034	0.0013	0.0014	0.0015	0.0019	0.0018
Attribute_type_change	0.0029	0.0021	0.0019	0.0014	0.0023	0.0022
Unclassified_change	0.0016	0.0013	0.0019	0.0020	0.0010	0.0020
Parent_class_change	0.0015	0.0026	0.0024	0.0014	0.0016	0.0009
Parent_interface_insert	0.0013	0.0041	0.0045	0.0009	0.0017	0.0011
Parameter_ordering_change	0.0013	0.0000	0.0000	0.0009	0.0005	0.0004
Parent_interface_delete	0.0010	0.0046	0.0045	0.0006	0.0012	0.0009
Additional_class	0.0010	0.0000	0.0014	0.0010	0.0018	0.0002
Removing_attribute_modifiability	0.0009	0.0013	0.0012	0.0007	0.0013	0.0012
Adding_attribute_modifiability	0.0008	0.0003	0.0005	0.0002	0.0003	0.0013
Removed_class	0.0006	0.0000	0.0026	0.0005	0.0006	0.0001
Adding_method_overridability	0.0005	0.0003	0.0002	0.0012	0.0002	0.0001
Removing_method_overridability	0.0005	0.0003	0.0002	0.0004	0.0003	0.0003
Return_type_insert	0.0004	0.0003	0.0002	0.0003	0.0004	0.0004
Return_type_delete	0.0004	0.0003	0.0000	0.0003	0.0001	0.0001
Removing_class_derivability	0.0003	0.0003	0.0005	0.0002	0.0002	0.0001
Parent_interface_change	0.0003	0.0005	0.0005	0.0002	0.0006	0.0006
Parent_class_insert	0.0003	0.0003	0.0002	0.0002	0.0002	0.0001
Parent_class_delete	0.0001	0.0000	0.0005	0.0001	0.0002	0.0001
Adding_class_derivability	0.0001	0.0000	0.0000	0.0001	0.0001	0.0000
Class_renaming	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Repair Size	1	2	3	4	5	6	
argouml	**5** (996)	**53** (638)	**315** (386)	**2682** (362)	**19975** (254)	∞(234)	∞(1
carol	**5** (30)	**8** (15)	**595** (10)	**3906** (10)	**3130** (7)	∞(13)	∞
columba	**6** (382)	**49** (255)	**417** (144)	**3522** (146)	**28118** (113)	**99766** (108)	∞
dnsjava	**6** (165)	**48** (139)	**402** (71)	**1990** (82)	**34961** (54)	∞(50)	∞
jEdit	**6** (115)	**46** (84)	**287** (53)	**3353** (48)	**27966** (32)	∞(30)	∞
jboss	**6** (514)	**48** (353)	**393** (208)	**3248** (189)	**26872** (147)	∞(150)	∞
jhotdraw6	**5** (21)	**37** (21)	**396** (9)	**517** (10)	**4769** (10)	**12428** (3)	**22242**
junit	**5** (40)	**45** (39)	**268** (18)	**95763** (11)	**10305** (7)	∞(11)	∞
log4j	**5** (223)	**45** (134)	**461** (68)	**2655** (70)	**13542** (64)	∞(42)	∞
org.eclipse.jdt.core	**5** (1606)	**40** (1025)	**305** (657)	**2318** (631)	**12427** (392)	∞(416)	∞(3
org.eclipse.ui.workbench	**5** (1184)	**53** (783)	**278** (414)	**2431** (464)	**15999** (326)	∞(305)	∞(2
scarab	**6** (653)	**44** (346)	**340** (202)	**3138** (159)	**9668** (113)	**84180** (137)	∞
struts	**5** (221)	**45** (133)	**284** (86)	**2686** (103)	**5862** (61)	**95470** (77)	∞
tomcat	**6** (281)	**46** (167)	**399** (111)	**3323** (120)	**19468** (84)	∞(87)	∞

Table A.7: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CT. The repair model CT is made from the distribution probability of changes included in 1-SC transaction bags.

Repair Size	1	2	3	4	5	6	
argouml	**6** (996)	**13** (638)	**86** (386)	**267** (362)	**1394** (254)	**5977** (234)	**16748** (19
carol	**7** (30)	**13** (15)	**121** (10)	**466** (10)	**494** (7)	**24117** (13)	**14019**
columba	**3** (382)	**13** (255)	**68** (144)	**552** (146)	**940** (113)	**2111** (108)	**10908** (7
dnsjava	**6** (165)	**13** (139)	**101** (71)	**218** (82)	**1553** (54)	**5063** (50)	**16363** (3
jEdit	**3** (115)	**13** (84)	**58** (53)	**251** (48)	**2906** (32)	**3189** (30)	**5648** (2
jboss	**6** (514)	**15** (353)	**88** (208)	**272** (189)	**1057** (147)	**6034** (150)	**13148** (8
jhotdraw6	**7** (21)	**13** (21)	**159** (9)	**187** (10)	**1779** (10)	**611** (3)	∞
junit	**3** (40)	**42** (39)	**596** (18)	∞(11)	**49345** (7)	∞(11)	**31634**
log4j	**6** (223)	**15** (134)	**146** (68)	**665** (70)	**6459** (64)	**16879** (42)	**55582** (4
org.eclipse.jdt.core	**6** (1606)	**26** (1025)	**93** (657)	**291** (631)	**1704** (392)	**4639** (416)	**18344** (31
org.eclipse.ui.workbench	**3** (1184)	**13** (783)	**74** (414)	**311** (464)	**1023** (326)	**6035** (305)	**22864** (21
scarab	**6** (653)	**16** (346)	**113** (202)	**420** (159)	**764** (113)	**3914** (137)	**13104** (8
struts	**3** (221)	**17** (133)	**100** (86)	**222** (103)	**675** (61)	**4785** (77)	**16796** (3
tomcat	**3** (281)	**13** (167)	**135** (111)	**431** (120)	**1068** (84)	**3497** (87)	**7407** (6

Table A.8: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CT. The repair model CT is made from the distribution probability of changes included in 5-SC transaction bags.

Repair Size	1	2	3	4	5	6	
argouml	**5** (996)	**20** (638)	**107** (386)	**482** (362)	**2160** (254)	**6439** (234)	**13733** (1
carol	**9** (30)	**20** (15)	**104** (10)	**350** (10)	**642** (7)	**12378** (13)	**8994**
columba	**4** (382)	**20** (255)	**105** (144)	**276** (146)	**1530** (113)	**1156** (108)	**5887**
dnsjava	**5** (165)	**20** (139)	**114** (71)	**552** (82)	**755** (54)	**3651** (50)	**9206**
jEdit	**4** (115)	**20** (84)	**81** (53)	**224** (48)	**1570** (32)	**1255** (30)	**2418**
jboss	**5** (514)	**20** (353)	**104** (208)	**337** (189)	**1029** (147)	**6188** (150)	**8602**
jhotdraw6	**9** (21)	**20** (21)	**228** (9)	**209** (10)	**3418** (10)	**279** (3)	**46060**
junit	**4** (40)	**37** (39)	**494** (18)	**92607** (11)	∞(7)	∞(11)	**50868**
log4j	**5** (223)	**20** (134)	**165** (68)	**611** (70)	**9125** (64)	**16351** (42)	**36244**
org.eclipse.jdt.core	**5** (1606)	**20** (1025)	**105** (657)	**416** (631)	**1587** (392)	**4680** (416)	**11829** (3
org.eclipse.ui.workbench	**4** (1184)	**19** (783)	**102** (414)	**327** (464)	**885** (326)	**4847** (305)	**9133** (2
scarab	**5** (653)	**20** (346)	**127** (202)	**555** (159)	**791** (113)	**2942** (137)	**12977**
struts	**4** (221)	**23** (133)	**106** (86)	**569** (103)	**1009** (61)	**8799** (77)	**7383**
tomcat	**4** (281)	**19** (167)	**140** (111)	**416** (120)	**853** (84)	**1297** (87)	**3215**

Table A.9: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CT. The repair model CT is made from the distribution probability of changes included in 10-SC transaction bags.

Repair Size	1	2	3	4	5	6	
argouml	**5** (996)	**29** (638)	**164** (386)	**391** (362)	**2183** (254)	**5672** (234)	**13597** (19
carol	**11** (30)	**27** (15)	**171** (10)	**639** (10)	**550** (7)	**10073** (13)	**9619**
columba	**4** (382)	**27** (255)	**153** (144)	**267** (146)	**1261** (113)	**903** (108)	**4451** (7
dnsjava	**4** (165)	**27** (139)	**169** (71)	**885** (82)	**720** (54)	**2453** (50)	**6610** (3
jEdit	**4** (115)	**27** (84)	**129** (53)	**202** (48)	**1153** (32)	**1368** (30)	**2745** (2
jboss	**4** (514)	**27** (353)	**165** (208)	**293** (189)	**855** (147)	**6132** (150)	**8689** (8
jhotdraw6	**11** (21)	**27** (21)	**162** (9)	**244** (10)	**6693** (10)	**190** (3)	**83682**
junit	**4** (40)	**33** (39)	**402** (18)	∞(11)	∞(7)	∞(11)	**26698**
log4j	**5** (223)	**28** (134)	**178** (68)	**1127** (70)	**12551** (64)	**20263** (42)	**19718** (4
org.eclipse.jdt.core	**4** (1606)	**27** (1025)	**165** (657)	**371** (631)	**1205** (392)	**4508** (416)	**9586** (31
org.eclipse.ui.workbench	**4** (1184)	**27** (783)	**126** (414)	**303** (464)	**781** (326)	**3925** (305)	**6665** (21
scarab	**4** (653)	**27** (346)	**167** (202)	**650** (159)	**729** (113)	**2698** (137)	**17857** (8
struts	**5** (221)	**28** (133)	**173** (86)	**439** (103)	**1095** (61)	**6348** (77)	**6682** (3
tomcat	**4** (281)	**26** (167)	**161** (111)	**410** (120)	**820** (84)	**1078** (87)	**2536** (6

Table A.10: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CT. The repair model CT is made from the distribution probability of changes included in 20-SC transaction bags.

Repair Size	1	2	3	4	5	6	
argouml	**6** (996)	**53** (638)	**147** (386)	**419** (362)	**2473** (254)	**6049** (234)	**12190** (19
carol	**17** (30)	**47** (15)	**390** (10)	**1841** (10)	**1484** (7)	**3803** (13)	**2962** (
columba	**6** (382)	**22** (255)	**134** (144)	**254** (146)	**697** (113)	**1380** (108)	**5304** (7
dnsjava	**6** (165)	**48** (139)	**392** (71)	**438** (82)	**818** (54)	**2781** (50)	**10734** (3
jEdit	**6** (115)	**46** (84)	**89** (53)	**183** (48)	**1269** (32)	**1616** (30)	**2020** (2
jboss	**6** (514)	**47** (353)	**253** (208)	**444** (189)	**1048** (147)	**8610** (150)	**5981** (8
jhotdraw6	**17** (21)	**47** (21)	**140** (9)	**498** (10)	**26388** (10)	**229** (3)	∞(
junit	**6** (40)	**47** (39)	**1160** (18)	∞(11)	∞(7)	**70696** (11)	**24882** (
log4j	**6** (223)	**47** (134)	**389** (68)	**2409** (70)	**48093** (64)	**8748** (42)	**92785** (4
org.eclipse.jdt.core	**6** (1606)	**43** (1025)	**211** (657)	**410** (631)	**1333** (392)	**5036** (416)	**10949** (31
org.eclipse.ui.workbench	**6** (1184)	**30** (783)	**141** (414)	**289** (464)	**742** (326)	**4778** (305)	**6896** (21
scarab	**6** (653)	**48** (346)	**163** (202)	**435** (159)	**718** (113)	**1998** (137)	**17669** (8
struts	**6** (221)	**47** (133)	**388** (86)	**301** (103)	**1238** (61)	**8103** (77)	**8900** (3
tomcat	**6** (281)	**18** (167)	**161** (111)	**399** (120)	**1008** (84)	**1035** (87)	**3494** (6

Table A.11: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CT. The repair model CT is made from the distribution probability of changes included in BFP transaction bags.

Repair Size	1	2	3	4	5	6	
argouml	**5** (996)	**37** (638)	**152** (386)	**417** (362)	**2436** (254)	**6661** (234)	**11243** (19
carol	**17** (30)	**32** (15)	**213** (10)	**1296** (10)	**954** (7)	**3026** (13)	**2918** (
columba	**5** (382)	**25** (255)	**139** (144)	**334** (146)	**688** (113)	**1582** (108)	**7843** (7
dnsjava	**5** (165)	**32** (139)	**213** (71)	**709** (82)	**1058** (54)	**3375** (50)	**13393** (3
jEdit	**5** (115)	**30** (84)	**75** (53)	**154** (48)	**2046** (32)	**2633** (30)	**4247** (2
jboss	**5** (514)	**31** (353)	**202** (208)	**506** (189)	**1054** (147)	**8775** (150)	**8844** (8
jhotdraw6	**17** (21)	**32** (21)	**95** (9)	**517** (10)	**9956** (10)	**348** (3)	∞(
junit	**5** (40)	**42** (39)	**935** (18)	∞(11)	∞(7)	**50700** (11)	**56637** (
log4j	**5** (223)	**32** (134)	**233** (68)	**1463** (70)	**29316** (64)	**5718** (42)	**33218** (4
org.eclipse.jdt.core	**5** (1606)	**31** (1025)	**204** (657)	**522** (631)	**1751** (392)	**6037** (416)	**16561** (31
org.eclipse.ui.workbench	**5** (1184)	**31** (783)	**127** (414)	**380** (464)	**1086** (326)	**4542** (305)	**9407** (21
scarab	**5** (653)	**31** (346)	**191** (202)	**652** (159)	**755** (113)	**1987** (137)	**17309** (8
struts	**5** (221)	**31** (133)	**211** (86)	**422** (103)	**1818** (61)	**9147** (77)	**5759** (3
tomcat	**5** (281)	**25** (167)	**190** (111)	**411** (120)	**1472** (84)	**1229** (87)	**4844** (6

Table A.12: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CT. The repair model CT is made from the distribution probability of changes included in ALL transaction bags.

Repair Size	1	2	3	4	5	6	7	
argouml	**9** (996)	**370** (638)	**13704** (386)	∞(362)	∞(254)	∞(234)	∞(197)	∞
carol	**8** (30)	**78** (15)	∞(10)	∞(10)	∞(7)	∞(13)	∞(6)	
columba	**8** (382)	**182** (255)	**8115** (144)	∞(146)	∞(113)	∞(108)	∞(73)	∞
dnsjava	**8** (165)	**1161** (139)	**38837** (71)	∞(82)	∞(54)	∞(50)	∞(33)	∞
jEdit	**8** (115)	**153** (84)	**70402** (53)	∞(48)	∞(32)	∞(30)	∞(29)	∞
jboss	**8** (514)	**1600** (353)	**14252** (208)	∞(189)	∞(147)	∞(150)	∞(86)	∞
jhotdraw6	**7** (21)	**62** (21)	**958** (9)	∞(10)	∞(10)	∞(3)	∞(5)	
junit	**8** (40)	**1759** (39)	∞(18)	∞(11)	∞(7)	∞(11)	∞(9)	
log4j	**7** (223)	**101** (134)	**4279** (68)	∞(70)	∞(64)	∞(42)	∞(41)	∞
org.eclipse.jdt.core	**8** (1606)	**2059** (1025)	**81214** (657)	∞(631)	∞(392)	∞(416)	∞(314)	∞
org.eclipse.ui.workbench	**9** (1184)	**361** (783)	**14231** (414)	∞(464)	∞(326)	∞(305)	∞(215)	∞
scarab	**8** (653)	**85** (346)	**4454** (202)	∞(159)	∞(113)	∞(137)	∞(89)	∞
struts	**7** (221)	**358** (133)	**9358** (86)	∞(103)	∞(61)	∞(77)	∞(39)	∞
tomcat	**7** (281)	**156** (167)	**14218** (111)	∞(120)	∞(84)	∞(87)	∞(61)	∞

Table A.13: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CTET. The repair model CTET is made from the distribution probability of changes included in 1-SC transaction bags.

Repair Size	1	2	3	4	5	6	7
argouml	**10** (996)	**208** (638)	**3043** (386)	**33309** (362)	∞(254)	∞(234)	∞(197)
carol	**11** (30)	**145** (15)	**8717** (10)	∞(10)	**58205** (7)	∞(13)	∞(6)
columba	**11** (382)	**151** (255)	**1995** (144)	**51215** (146)	∞(113)	∞(108)	∞(73)
dnsjava	**11** (165)	**408** (139)	**4217** (71)	∞(82)	∞(54)	∞(50)	∞(33)
jEdit	**11** (115)	**144** (84)	**3090** (53)	**23302** (48)	∞(32)	∞(30)	∞(29)
jboss	**11** (514)	**290** (353)	**3267** (208)	**92063** (189)	∞(147)	∞(150)	∞(86)
jhotdraw6	**10** (21)	**118** (21)	**880** (9)	**22708** (10)	∞(10)	∞(3)	∞(5)
junit	**11** (40)	**1285** (39)	**4353** (18)	∞(11)	∞(7)	∞(11)	∞(9)
log4j	**7** (223)	**124** (134)	**1385** (68)	**29454** (70)	∞(64)	∞(42)	∞(41)
org.eclipse.jdt.core	**13** (1606)	**274** (1025)	**5154** (657)	**74267** (631)	∞(392)	∞(416)	∞(314)
org.eclipse.ui.workbench	**9** (1184)	**180** (783)	**1879** (414)	**32900** (464)	∞(326)	∞(305)	∞(215)
scarab	**10** (653)	**126** (346)	**1318** (202)	**22650** (159)	∞(113)	∞(137)	∞(89)
struts	**11** (221)	**218** (133)	**2887** (86)	**47203** (103)	∞(61)	∞(77)	∞(39)
tomcat	**10** (281)	**160** (167)	**2129** (111)	**23455** (120)	∞(84)	∞(87)	∞(61)

Table A.14: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CTET. The repair model CTET is made from the distribution probability of changes included in 5-SC transaction bags.

Repair Size	1	2	3	4	5	6	7
argouml	**13** (996)	**199** (638)	**3809** (386)	**34189** (362)	∞(254)	∞(234)	∞(197)
carol	**14** (30)	**217** (15)	**5863** (10)	∞(10)	**59433** (7)	∞(13)	∞(6)
columba	**14** (382)	**169** (255)	**2301** (144)	**40661** (146)	∞(113)	∞(108)	∞(73)
dnsjava	**14** (165)	**331** (139)	**4509** (71)	**76515** (82)	∞(54)	∞(50)	∞(33)
jEdit	**14** (115)	**175** (84)	**3754** (53)	**20256** (48)	∞(32)	∞(30)	∞(29)
jboss	**14** (514)	**270** (353)	**2782** (208)	**64260** (189)	∞(147)	∞(150)	∞(86)
jhotdraw6	**13** (21)	**115** (21)	**1350** (9)	**33871** (10)	∞(10)	∞(3)	∞(5)
junit	**14** (40)	**1060** (39)	**4472** (18)	∞(11)	∞(7)	∞(11)	∞(9)
log4j	**9** (223)	**152** (134)	**1611** (68)	**29279** (70)	∞(64)	∞(42)	∞(41)
org.eclipse.jdt.core	**17** (1606)	**296** (1025)	**4594** (657)	**58172** (631)	∞(392)	∞(416)	∞(314)
org.eclipse.ui.workbench	**12** (1184)	**185** (783)	**1989** (414)	**26960** (464)	∞(326)	∞(305)	∞(215)
scarab	**13** (653)	**139** (346)	**1581** (202)	**20599** (159)	∞(113)	∞(137)	∞(89)
struts	**14** (221)	**262** (133)	**4067** (86)	**45897** (103)	∞(61)	∞(77)	∞(39)
tomcat	**13** (281)	**178** (167)	**2499** (111)	**17274** (120)	∞(84)	∞(87)	∞(61)

Table A.15: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CTET. The repair model CTET is made from the distribution probability of changes included in 10-SC transaction bags.

Repair Size	1	2	3	4	5	6	7
argouml	**16** (996)	**262** (638)	**3869** (386)	**39846** (362)	∞(254)	∞(234)	∞(197)
carol	**16** (30)	**230** (15)	**9186** (10)	∞(10)	**70501** (7)	∞(13)	∞(6)
columba	**16** (382)	**184** (255)	**2759** (144)	**34348** (146)	∞(113)	∞(108)	∞(73)
dnsjava	**16** (165)	**359** (139)	**5383** (71)	**73244** (82)	∞(54)	∞(50)	∞(33)
jEdit	**16** (115)	**183** (84)	**3393** (53)	**24577** (48)	∞(32)	∞(30)	∞(29)
jboss	**15** (514)	**327** (353)	**3238** (208)	**62760** (189)	∞(147)	∞(150)	∞(86)
jhotdraw6	**16** (21)	**156** (21)	**2171** (9)	**50523** (10)	∞(10)	∞(3)	∞(5)
junit	**16** (40)	**1000** (39)	**5181** (18)	∞(11)	∞(7)	∞(11)	∞(9)
log4j	**11** (223)	**183** (134)	**2575** (68)	**39868** (70)	∞(64)	∞(42)	∞(41)
org.eclipse.jdt.core	**21** (1606)	**382** (1025)	**5057** (657)	**55975** (631)	∞(392)	∞(416)	∞(314)
org.eclipse.ui.workbench	**15** (1184)	**182** (783)	**2441** (414)	**30919** (464)	∞(326)	∞(305)	∞(215)
scarab	**15** (653)	**173** (346)	**2270** (202)	**33705** (159)	∞(113)	∞(137)	∞(89)
struts	**16** (221)	**339** (133)	**3364** (86)	**49330** (103)	∞(61)	∞(77)	∞(39)
tomcat	**15** (281)	**187** (167)	**2394** (111)	**22010** (120)	∞(84)	∞(87)	∞(61)

Table A.16: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CTET. The repair model CTET is made from the distribution probability of changes included in 20-SC transaction bags.

Repair Size	1	2	3	4	5	6	7
argouml	**19** (996)	**407** (638)	**6487** (386)	**99947** (362)	∞(254)	∞(234)	∞(197)
carol	**25** (30)	**417** (15)	**11467** (10)	∞(10)	∞(7)	∞(13)	∞(6)
columba	**17** (382)	**237** (255)	**4376** (144)	**51308** (146)	∞(113)	∞(108)	∞(73)
dnsjava	**25** (165)	**508** (139)	**7825** (71)	∞(82)	∞(54)	∞(50)	∞(33)
jEdit	**24** (115)	**265** (84)	**4044** (53)	**34097** (48)	∞(32)	∞(30)	∞(29)
jboss	**25** (514)	**422** (353)	**6031** (208)	∞(189)	∞(147)	∞(150)	∞(86)
jhotdraw6	**19** (21)	**423** (21)	**5741** (9)	**94185** (10)	∞(10)	∞(3)	∞(5)
junit	**25** (40)	**1213** (39)	**9622** (18)	∞(11)	∞(7)	∞(11)	∞(9)
log4j	**17** (223)	**352** (134)	**5801** (68)	**80747** (70)	∞(64)	∞(42)	∞(41)
org.eclipse.jdt.core	**31** (1606)	**414** (1025)	**7814** (657)	**86521** (631)	∞(392)	∞(416)	∞(314)
org.eclipse.ui.workbench	**24** (1184)	**278** (783)	**4583** (414)	**47871** (464)	∞(326)	∞(305)	∞(215)
scarab	**17** (653)	**274** (346)	**4646** (202)	**59593** (159)	∞(113)	∞(137)	∞(89)
struts	**24** (221)	**500** (133)	**8799** (86)	**86343** (103)	∞(61)	∞(77)	∞(39)
tomcat	**18** (281)	**340** (167)	**4113** (111)	**33663** (120)	∞(84)	∞(87)	∞(61)

Table A.17: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CTET. The repair model CTET is made from the distribution probability of changes included in BFP transaction bags.

Repair Size	1	2	3	4	5	6	7
argouml	**19** (996)	**364** (638)	**5749** (386)	**66875** (362)	∞(254)	∞(234)	∞(197)
carol	**21** (30)	**410** (15)	**14905** (10)	∞(10)	∞(7)	∞(13)	∞(6)
columba	**17** (382)	**257** (255)	**3770** (144)	**51588** (146)	∞(113)	∞(108)	∞(73)
dnsjava	**20** (165)	**508** (139)	**7936** (71)	∞(82)	∞(54)	∞(50)	∞(33)
jEdit	**19** (115)	**281** (84)	**4294** (53)	**40013** (48)	∞(32)	∞(30)	∞(29)
jboss	**20** (514)	**432** (353)	**5976** (208)	∞(189)	∞(147)	∞(150)	∞(86)
jhotdraw6	**19** (21)	**400** (21)	**4379** (9)	**75119** (10)	∞(10)	∞(3)	∞(5)
junit	**20** (40)	**985** (39)	**7228** (18)	∞(11)	∞(7)	∞(11)	∞(9)
log4j	**17** (223)	**291** (134)	**5843** (68)	**74260** (70)	∞(64)	∞(42)	∞(41)
org.eclipse.jdt.core	**25** (1606)	**375** (1025)	**8049** (657)	**96672** (631)	∞(392)	∞(416)	∞(314)
org.eclipse.ui.workbench	**20** (1184)	**288** (783)	**3985** (414)	**42118** (464)	∞(326)	∞(305)	∞(215)
scarab	**17** (653)	**277** (346)	**4347** (202)	**46263** (159)	∞(113)	∞(137)	∞(89)
struts	**20** (221)	**436** (133)	**6330** (86)	**83370** (103)	∞(61)	∞(77)	∞(39)
tomcat	**17** (281)	**301** (167)	**3466** (111)	**31254** (120)	∞(84)	∞(87)	∞(61)

Table A.18: The median number of attempts(in bold) required to find the correct repair shape of fix transactions. The values in brackets indicate the number of fix transactions tested per project and per transaction size for repair model CTET. The repair model CTET is made from the distribution probability of changes included in ALL transaction bags.

Appendix B

Measuring Software Redundancy

B.1 Dataset

Table B.1 presents the first and last commit analyzed in our experiments.

App	Commit Analyzed		Date
log4	First	bf15d7fe3214cbd4be1c2b998de915e7f801efc6	2000-11-16
	Last	07576bfa83e14bc856d423dc6f469467dede6f75	2013-05-13
junit	First	b6a0693454ac8ded32b3a1ea7c859c5a840169dc	2000-12-03
	Last	da0a727ebfcfee036d760d52f574f7b07aba7df2	2013-10-24
pico	First	07d3c80dcfb199ca66e4efdacfa0f6c756c07434	2010-02-24
	Last	71fe6c004d043650996febb1f2ff33078e3f7dc4	2013-07-05
collections	First	6aff6757955abc3fc11ed3f1ee10e6e1d8351e34	2001-04-14
	Last	77180fc617907659d2f7cf21cc7f15a74d4f3448	2013-06-30
math	First	a859606a95d3eca7d8e13e8fff11daa5b1a40813	2003-05-12
	Last	de4209544270def43e39db0d214d1564939f8e40	2013-07-08
lang	First	fe460cb485a9d224b37a521b0ccef616ff786876	2002-07-19
	Last	2c454a4ce3fe771098746879b166ede2284b94f4	2013-07-08

Table B.1: Project Features.

B.2 Temporal Redundant commits

We present the temporal redundant commits for each project.

Commons collections

- Redundant commits: Global - line
- Redundant commits: Global - token
- Redundant commits: Local - line
- Redundant commits: Local - token

Commons lang

- Redundant commits: Global - line


Redundant commits: Global - token
Redundant commits: Local - line
Redundant commits: Local - token

Commons math
Redundant commits: Global - line
Redundant commits: Global - token
Redundant commits: Local - line
Redundant commits: Local - token

Pico
Redundant commits: Global - line
Redundant commits: Global - token
Redundant commits: Local - line
Redundant commits: Local - token

jUnit
Redundant commits: Global - line
Redundant commits: Global - token
Redundant commits: Local - line
Redundant commits: Local - token

log4j
Redundant commits: Global - line
Redundant commits: Global - token
Redundant commits: Local - line
Redundant commits: Local - token

Bibliography

[1] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, (Piscataway, NJ, USA), pp. 3–13, IEEE Press, 2012.

[2] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 550–554, IEEE Computer Society, 2009.

[3] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," pp. 87–102, 2009.

[4] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, (Washington, DC, USA), pp. 392–395, IEEE Computer Society, 2011.

[5] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 802–811, IEEE Press, 2013.

[6] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for web applications," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, (New York, NY, USA), pp. 237–246, ACM, 2010.

[7] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Engg.*, vol. 10, pp. 405–435, Oct. 2005.

[8] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, (Piscataway, NJ, USA), pp. 3–13, IEEE Press, 2012.

[9] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, pp. 286–315, June 2009.

[10] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 772–781, IEEE Press, 2013.

[11] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Engineering*, pp. 1–30, 2013.

[12] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 364–374, IEEE Computer Society, 2009.

[13] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, (Washington, DC, USA), pp. 65–74, IEEE Computer Society, 2010.

[14] E. W. Myers, "An o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[15] Z. Xing and E. Stroulia, "Umldiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, (New York, NY, USA), pp. 54–65, ACM, 2005.

[16] J. I. Maletic and M. L. Collard, "Supporting source code difference analysis," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, (Washington, DC, USA), pp. 210–219, IEEE Computer Society, 2004.

[17] T. Zimmermann, "Fine-grained processing of cvs archives with apfel," in *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '06, (New York, NY, USA), pp. 16–20, ACM, 2006.

[18] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: a semantic-graph differencing tool for studying changes in large code bases," in *20th IEEE International Conference on Software Maintenance*, 2004.

[19] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," vol. 30, (New York, NY, USA), pp. 1–5, ACM, May 2005.

[20] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, pp. 725 –743, nov. 2007.

[21] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting similar java classes using tree algorithms," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, (New York, NY, USA), pp. 65–71, ACM, 2006.

[22] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, (Washington, DC, USA), pp. 35–45, IEEE Computer Society, 2006.

[23] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 180–190, Nov 2013.

[24] B. Fluri, E. Giger, and H. C. Gall, "Discovering patterns of change types," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, (Washington, DC, USA), pp. 463–466, IEEE Computer Society, 2008.

[25] B. Livshits and T. Zimmermann, "Dynamine: Finding common error patterns by mining software revision histories," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, (New York, NY, USA), pp. 296–305, ACM, 2005.

[26] S. Negara, M. Vakilian, N. Chen, R. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?," in *ECOOP 2012 – Object-Oriented Programming* (J. Noble, ed.), vol. 7313 of *Lecture Notes in Computer Science*, pp. 79–103, Springer Berlin Heidelberg, 2012.

[27] R. Robbes, M. Lanza, and M. Lungu, "An approach to software evolution based on semantic change," in *In FASE '07: Proceedings of the 10th Conference on Fundamental Approaches to Software Engineering*, pp. 27–41, 2007.

[28] R. Robbes, "Mining a change-based software repository," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, (Washington, DC, USA), pp. 15–, IEEE Computer Society, 2007.

[29] R. Robbes, *Of Change and Software*. PhD thesis, University of Lugano, 2008.

[30] T. Mens and T. Tourwe, "A declarative evolution framework for object-oriented design patterns," in *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pp. 570–579, 2001.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[32] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, (Washington, DC, USA), pp. 86–, IEEE Computer Society, 1995.

[33] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[34] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, 2005.

[35] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[36] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, (New York, NY, USA), pp. 147–156, ACM, 2010.

[37] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, (Piscataway, NJ, USA), pp. 837–847, IEEE Press, 2012.

[38] A. Alali, H. Kagdi, and J. Maletic, "What's a typical commit? a characterization of open source software repositories," in *IEEE International Conference on Program Comprehension*, 2008.

[39] L. Hattori and M. Lanza, "On the nature of commits," in *Automated Software Engineering - Workshops*, pp. 63 –71, sept. 2008.

[40] D. M. German, "An empirical study of fine-grained software modifications," *Empirical Softw. Engineering*, vol. 11, pp. 369–393, Sept. 2006.

[41] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *Proceedings of the International Working Conference on Mining Software Repositories*, 2008.

[42] A. Hindle, D. German, M. Godfrey, and R. Holt, "Automatic classication of large changes into maintenance categories," in *International Conference on Program Comprehension*, 2009.

[43] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," pp. 70–79, 2007.

[44] B. Fluri, M. Würsch, E. Giger, and H. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.

[45] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on*, pp. 120–130, 2000.

[46] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of the International Conference on Software Maintenance*, ICSM '03, (Washington, DC, USA), pp. 23–, IEEE Computer Society, 2003.

[47] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, (New York, NY, USA), pp. 121–130, ACM, 2009.

[48] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, (New York, NY, USA), pp. 23:304–23:318, ACM, 2008.

[49] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: Recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 15–25, ACM, 2011.

[50] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli, "A machine learning approach for text categorization of fixing-issue commits on cvs," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2010.

[51] A. Sureka, S. Lal, and L. Agarwal, "Applying fellegi-sunter (fs) model for traceability link recovery between bug databases and version archives," in *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, APSEC '11, (Washington, DC, USA), pp. 146–153, IEEE Computer Society, 2011.

[52] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–5, May 2005.

[53] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, (Washington, DC, USA), pp. 81–90, IEEE Computer Society, 2006.

[54] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr., "Mining version archives for co-changed lines," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, (New York, NY, USA), pp. 72–75, ACM, 2006.

[55] R. Purushothaman and D. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, pp. 511 – 526, june 2005.

[56] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *Principles of Software Evolution, Eighth International Workshop on*, pp. 13–22, Sept 2005.

[57] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Softw.*, vol. 26, pp. 26–33, Jan. 2009.

[58] J. Bevan, E. J. Whitehead, S. Kim, and M. Godfrey, "Facilitating software evolution research with kenyon," *ACM SIGSOFT Software Engineering Notes*, vol. 30, p. 177, Sept. 2005.

[59] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, pp. 446–465, June 2005.

[60] A. M. Daniel German, "Automating the measurement of open source projects," in *In Proceedings of the 3rd Workshop on Open Source Software Engineering*, pp. 63–67, 2003.

[61] D. E. Knuth, "The errors of tex," *Softw., Pract. Exper.*, vol. 19, no. 7, pp. 607–685, 1989.

[62] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification - a concept for in-process measurements," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 943–956, 1992.

[63] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *Journal of Systems and Software*, vol. 4, no. 4, pp. 289–300, 1984.

[64] S. K. Nath, R. Merkel, and M. F. Lau, "On the improvement of a fault classification scheme with implications for white-box testing," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, (New York, NY, USA), pp. 1123–1130, ACM, 2012.

[65] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, (New York, NY, USA), pp. 35–45, ACM, 2006.

[66] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 315–324, ACM, 2010.

[67] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation0," *Commun. ACM*, vol. 27, pp. 42–52, Jan. 1984.

[68] R. A. Demillo and A. P. Mathur, "A grammar based fault classification scheme and its application to the classification of the errors of tex," tech. rep., Software Engineering Research Center, Purdue University, 1995.

[69] A. Arcuri, "Evolutionary repair of faulty software," *Appl. Soft Comput.*, vol. 11, pp. 3494–3514, June 2011.

[70] M. Harman, U. Ph, and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, pp. 833–839, 2001.

[71] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[72] *The Unified Modeling Language User Guide*. Pearson Education, 2005.

[73] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, pp. 40–51, Oct 1992.

[74] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, (New York, NY, USA), pp. 61–72, ACM, 2010.

[75] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 947–954, ACM, 2009.

[76] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 254–265, ACM, 2014.

[77] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 550–554, IEEE Computer Society, 2009.

[78] W. Weimer, Z. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 356–366, Nov 2013.

[79] X. Mao, Y. Lei, and Y. Qi, "Making automatic repair for large-scale programs more efficient using weak recompilation," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, (Washington, DC, USA), pp. 254–263, IEEE Computer Society, 2012.

[80] Y. Qi, X. Mao, Y. Wen, Z. Dai, and B. Gu, "More efficient automatic repair of large-scale programs using weak recompilation," *Science China Information Sciences*, vol. 55, no. 12, pp. 2785–2799, 2012.

[81] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, (Washington, DC, USA), pp. 180–189, IEEE Computer Society, 2013.

[82] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, (New York, NY, USA), pp. 467–477, ACM, 2002.

[83] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, (New York, NY, USA), pp. 273–282, ACM, 2005.

[84] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, (Washington, DC, USA), pp. 27–, IEEE Computer Society, 2007.

[85] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, pp. 92–106, Dec. 2004.

[86] S. S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques.," in *ESEM* (H. D. Rombach, S. G. Elbaum, and J. Münch, eds.), pp. 41–50, ACM, 2008.

[87] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[88] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz, "Begbunch: Benchmarking for c bug detection tools," in *ISSTA 2009*, DEFECTS '09, (New York, USA), pp. 16–20, ACM, 2009.

[89] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, (New York, NY, USA), pp. 433–436, ACM, 2007.

[90] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with ast analysis," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 388–391, Sept 2013.

[91] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, (New York, NY, USA), pp. 83–92, ACM, 2011.

[92] M. Monperrus and M. Martinez, "CVS-Vintage: A Dataset of 14 CVS Repositories of Java Software," tech. rep.

[93] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 15, no. 1, pp. pp. 72–101, 1904.

[94] F. Galton, "Regression towards mediocrity in hereditary stature.," *Journal of the Anthropological Institute of Great Britain and Ireland*, pp. 246–263, 1886.

[95] D. of Mathematics of the University of York, "Statistical tables." `http://www.york.ac.uk/depts/maths/tables/`, Last visited: April 9 2013.

[96] J. Cohen *et al.*, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[97] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data.," *Biometrics*, vol. 33, pp. 159–174, Mar. 1977.

[98] F. L. Joseph, "Measuring nominal scale agreement among many raters," *Psychological bulletin*, vol. 76, no. 5, pp. 378–382, 1971.

[99] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[100] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, (New York, NY, USA), pp. 492–495, ACM, 2014.

[101] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), pp. 1–10, ACM, 2011.

[102] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Softw. Maint. Evol.*, vol. 19, pp. 77–131, Mar. 2007.

[103] M. Monperrus, "A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 234–242, ACM, 2014.

[104] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, (New York, NY, USA), pp. 30–39, ACM, 2014.

[105] V. Debroy and W. E. Wong, "Combining mutation and fault localization for automated program debugging," *Journal of Systems and Software*, vol. 90, no. 0, pp. 45 – 60, 2014.

[106] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, (New York, NY, USA), pp. 191–201, ACM, 2013.

[107] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, (Washington, DC, USA), pp. 39–46, IEEE Computer Society, 2006.

[108] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[109] Y. Ledru, A. Petrenko, and S. Boroday, "Using string distances for test case prioritisation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 510–514, IEEE Computer Society, 2009.

[110] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Automated Software Engg.*, vol. 19, pp. 65–95, Mar. 2012.

[111] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 159–182, Feb. 2002.

[112] M. Bóna, *A Walk Through Combinatorics: An Introduction to Enumeration and Graph Theory*. World Scientific, 2011.